Departmental Coversheet

Michaelmas Term 2019-20

Compilers Assignment

Candidate Number: 1035502
Your degree: Computer Science

1 Array loops

Implementing array loops (which we will also call range-based for loops) is fairly straightforward. The plan is to have the loop define a variable of the correct type that only exists within its scope. This variable will behave a lot like `var` parameters in the sense that its type is `T` while internally it is storing a pointer to `T`. Then, to loop over the array we just need to do pointer arithmetic in a C-style way.

1.1 Abstract syntax

We create a new type of `stmt_guts` for this new array loop. Notice the two slots provided for storing the iterator variable and the upper bound.

```
and stmt_guts = ...
  | RangeStmt of ident * expr * stmt * def option ref * def
      option ref
```

We also enhance the pretty print to support this type of statement, which is trivial. Note that we do not print the `def`s, as they are only used internally.

1.2 Lexer

The lexer only needs to be extended with the keyword `in`, as `for` is already a keyword. Notice that this breaks the `strcopy.p` test that comes with the compiler because it uses `in` as a variable name. The fix is to simply rename the variable in the test.

1.3 Parser

The parser needs to be extended with a new `token` type – `IN`. After that we need to add new production rules for generating the abstract syntax out of the concrete one:

```
stmt1 : ...
  | FOR IDENT IN variable DO stmts END  { RangeStmt ($2, $4,
      $6, ref None, ref None) }
```

Notice that we are using `IDENT` for the iterator variable, this follows the pattern of other declarations. Also, we are using `variable` for the array. This is because picoPascal does not seem to support returning arrays and thus any expression whose type is an array type must be a variable at compile time.

1.4 Semantic analysis

The first thing we need to do before starting the actual semantic analysis is to add a new `def_kind` type that we use for the iterator variable:

```
type def_kind = ...
  | RefDef                          (* Reference *)
```

Now comes the more interesting part, namely the actual semantic analysis. To verify that a `RangeStmt` is semantically correct we just need to verify that the expression that specifies the array is correct and that its type is an array.

From that we can deduce the type of the iterator variable. We then need to create its definition and allocate it. After that we do the same for the upper bound variable. Finally, we update the local environment to contain the iterator definition and semantically check the body of the loop using that environment.

```
let rec check_stmt s env alloc =
  err_line := s.s_line;
  match s.s_guts with ...
    | RangeStmt (x, arr, body, iter_ref, upb_ref) ->
        let at = check_expr arr env in
        begin
          match at.t_guts with
              ArrayType (_, bt) ->
                let iter = make_def x RefDef bt in
                alloc iter; iter_ref := Some iter;
                let upb = make_def (intern "*iter_upb*")
                    RefDef bt in
                alloc upb; upb_ref := Some upb;
                let env' = add_def iter env in
                check_stmt body env' alloc
            | _ ->
                sem_error "range-based for loop over non-array
                    " []
```

We also need to modify the allocation procedure to properly support references; they need to use the representation of `addrtype`. A utility `is_referece` function is added to support this:

```
let local_alloc size nreg d =
  if !regvars && not d.d_mem && (scalar d.d_type ||
      is_reference d) && !nreg < Mach.nregvars then begin
    d.d_addr <- Register !nreg; incr nreg
  end
  else begin
    let r = if is_reference d then addrtype.t_rep else d.
        d_type.t_rep in
    align r.r_align size;
    size := !size + r.r_size;
    d.d_addr <- Local (local_base - !size)
  end
```

We had to change both the `if` condition and the `else` body. The change of the `if` condition allows us to store references in registers, which is quite important for efficiency. The change of the `else` body is to use the correct representation for references, if it is not possible to store them in a register.

We also add `RefDef` to the list accepted constructors in `do_alloc`. This allows array loops to work in functions and procedures.

1.5 Code generation

Generating code for this new construct follows a similar pattern to other loops. The only new thing is the pointer arithmetic for quicker iteration.

```
let rec gen_stmt s =
  let code =
    match s.s_guts with ...
      | RangeStmt (_, arr, body, iter_ref, upb_ref) ->
          let iter = match !iter_ref with Some d -> d | _ ->
              failwith "range for"
          and upb = match !upb_ref with Some d -> d | _ ->
              failwith "range for" in
          let l1 = label () and l2 = label () in
          <SEQ,
            <STOREW, gen_addr arr, address iter>,
            <STOREW,
              <OFFSET, <LOADW, address iter>, <CONST arr.
                  e_type.t_rep.r_size>>,
              address upb>,
            <JUMP l2>,
            <LABEL l1>,
            gen_stmt body,
            <STOREW,
              <OFFSET, <LOADW, address iter>, <CONST (
                  base_type arr.e_type).t_rep.r_size>>,
              address iter>,
            <LABEL l2>,
            <JUMPC (Neq, l1), <LOADW, address iter>, <LOADW,
                address upb>>>
```

The labels `l1` and `l2` correspond to the start and end of the loop body respectively. The code before `l1` initializes `iter` with the address of the first element and `upb` – with the address one element past the end of the array. We do not need to do multiplication, as we already know the total size of the array.

We use the standard `while` loop trick of having only one jump per iteration. The body consists of the statements' code followed by incrementing the iterator by the size of a single element. This is executed until `iter` reaches `upb`.

To access the raw pointers of the references we use the `address` function. However, the code in the body treats the iterator like a regular variable and uses the `gen_addr` function. So, to support these operations we need to modify it. This only involves changing one of the cases to include `RefDef`:

```
let rec gen_addr v =
  match v.e_guts with
      Variable x ->
        let d = get_def x in
        begin
          match d.d_kind with ...
            | RefDef | VParamDef ->
                <LOADW, address d>
```

1.6 Testing

To test the efficiency of our implementation and the new construct in general, we test it against the naive approach of looping over the indices with a traditional `for` loop. We compare the number of instructions for the two approaches with varying degrees of optimization:

Test name	Ø	-O	-O2
`range.p`	28	28	26
`range_for.p`	46	44	34
`range_record.p`	47	47	43
`range_record_for.p`	77	73	55
`range_multidim.p`	67	67	62
`range_multidim_for.p`	91	89	75
`range_pointer.p`	43	43	40
`range_pointer_for.p`	67	64	51
`range_pointer2.p`	44	44	40
`range_pointer2_for.p`	62	60	48
`range_ref.p`	28	25	26
`range_ref_for.p`	34	29	30

As seen by the results, the array loop performs strictly better on all six tests. For reference, here is a summary of what each test consists of:

- `range.p` – Fill an array with the integers 0 to 9, then print its contents.
- `range_record.p` – The provided test, which uses an array of records.
- `range_multidim.p` – Fill a 2D integer array such that $a_{i,j} = i + j$, then print a table with its contents.
- `range_pointer.p` – Fill an array of dynamically allocated integers with the values 0 to 9 by first calling new on each element, then sum up its contents and print the sum.
- `range_pointer2.p` – Similar to the last one, except we have a pointer to array instead of an array with pointers.
- `range_ref.p` – Similar to the last one, except we pass the array as a `var` parameter to an `init` procedure, which fills the array, and then a `find_sum` function, which finds the sum.

Furthermore, all of the previous tests pass, as the changes do not affect them in any way, the exception being `strcopy.p`, which uses `in` as a variable name, as mentioned above.

2 Commuting code

The main idea behind our approach for this task is to compute how each piece of code affects the mutable state and/or the input and output. After we have that, it is not too difficult to check whether two pieces of code commute. As will be seen, our approach handles all language features and common constructs quite well.

As for the implementation, we have chosen to separate this "effect" analysis from the semantic analysis and thus most of the code is located in the newly created file `com_check.ml`. Now, let us first cover the trivial parts of the implementation.

2.1 Abstract syntax

A new type of `stmt_guts` was introduced to the abstract syntax:

```
and stmt_guts = ...
  | Commutative of stmt * stmt
```

The pretty print was also enhanced to support this type of statement.

2.2 Lexer

No additions are needed to the lexer because it already recognizes all of the relevant tokens.

2.3 Parser

The parser just needs to be extended with one new production rule:

```
stmt1 : ...
  | SUB stmts COLON stmts BUS          { Commutative ($2, $4) }
```

2.4 Semantic analysis

The semantic analysis for this new statement is also trivial. It is treated the same way a two element `Seq` statement would be:

```
let rec check_stmt s env alloc =
  err_line := s.s_line;
  match s.s_guts with ...
    | Commutative (s1, s2) ->
        check_stmt s1 env alloc;
        check_stmt s2 env alloc
```

2.5 The main file

In the main file instead of calling `Tgen.translate` after we are done with the semantic analysis we do the following:

```
(* Effect analysis *)
begin try Com_check.com_check prog with
  Com_check.Com_error str ->
    printf "Possibly incorrect\n" [];
    exit 0
end;
printf "Correct\n" [];
exit 0
```

Note that the new `Com_error` exception comes with a description of the error. Here we purposefully ignore it because of the specification of the task. (It was however used during the development stage.)

Also, a new compiler flag, whose purpose will be explained further below, is added to the beginning:

```
let spec =
  Arg.align
    [ ...
      "-strictio", Arg.Unit (fun () -> Com_check.strict_io :=
          true),
        " take stricter approach to io commutativity"]
```

2.6 Effect analysis

Before going into the implementation, let us first discuss what it means for two pieces of code to commute. More specifically, we are interested how to treat the io with respect to code that can produce errors or get stuck in an infinite loop. We can take one of three stances:

- We assume the code is correct or rather we do not care what exactly happens, if it throws an error or gets stuck. We only care about the sequence of io events that happens assuming no errors.
- We care about exactly when each piece of io happens (this includes throwing errors). We want to get the exact same results with the two orderings of the commuting code.
- Something inbetween the two stances above: exact times are not important but we do want all io events to happen in the same order even if there are errors.

We claim that the third view is not very practical. Consider a program which consists of two bits of code: a print statement and a “wait ten million years” statement. This approach says that they commute, but if the second one was a "wait forever" statement, they would not commute. However, in practice both are effectively the same and thus this view is not at all useful.

Our version of the compiler supports both the first and the second option. By default, it uses the first one but we can switch to the stricter view with the `-strictio` flag that was introduced. We will mostly focus on the lax view in this report, as it is more interesting and useful.

2.6.1 The model

We model what effects each piece of code could have. With the exception of several flags and other similar things we only consider how it affects primitive types and pointers. All other types are made up of these building blocks and thus all effects on them can be decomposed into the ones described above.

For each object of interest we track whether it could be read (have its value accessed) and whether it will/can be written to (have its value overwritten). This is represented as a pair:

```
type state = Same | Overwritten | Partial | Invalid
type effect = bool * state
```

The boolean flag indicates whether the object could be read. The `state` represents what state the object is left in after the operation. The meanings of the four different `state`s are as follows: its value is left unchanged; its value is definitely overwritten; its value may or may not be overwritten (or for some objects – could be partially modified); it is left in an undefined state and cannot be accessed (doing so would be an error). As we proceed, it will become obvious why we have these four different states.

Of course, this only specifies how one single object is affected. Most programs have many such objects and we need some way to identify them. It is not practical to just use the `d_tag` of their definitions, as different sorts of objects may need different treatment. Furthermore, multiple objects may be associated with a single definition (and some might be associated with no definition). This is why we introduce the following type:

```
type id_type =
    Var of int * int * ident
  | Ref of int * int * ident
  | Heap of int
  | Element of int * id_type
  | Field of int * ident * id_type
  | Return
  | IO
```

All `id_type`s (with the exception of `Return` and `IO`, which are used as special indicators) first store their type in the form of a type id – the `t_id` field of the `ptype`. Then, for `Var` and `Ref`, which refer to actual definitions in the code, we store their level and identifier. `Heap` objects are stored in dynamically allocated memory and are found by pointers that can change. Because of that we can never be sure (in the general case) which is which, so we only store their type. Finally, to enable us to only store the primitive types we use `Element` and `Field` to break down compound objects into their composite parts. While `Field` objects store which field of a record they refer to, as this is known in compile time, `Element` objects do not store which element of an array they refer to because the index is not usually known in compile time. This means that a single `Element` object represents all elements of the array. As mentioned, `Return` and `IO` are special indicator objects that we use to simplify our design. Their exact purpose will be explained later.

All of these `effect`s and `id`s are stored in a `Map` from `id_type` to `effect`. Here the approach for the map from the `dict.ml` file was taken:

```
module IdMap = Map.Make(struct
    type t = id_type
    let compare = compare
  end)

type effect_map = effect IdMap.t
```

However, for the effect of an expression/statement we also need a few extra flags and lists, so we define the following type:

```
type big_effect = bool * bool * effect_map * (int list) * ((
    id_type * id_type) list)
```

This type is intentionally implemented as a product type instead of a record. This is because in virtually all of its uses it needs to be deconstructed and potentially constructed again. Therefore, using a simple product type avoids having overly long constructions. The importance of each of its components will be understood later but here is a quick rundown of its elements (there are comments that explain them in the actual code):

- an "is noncommutative" flag – used only for handling of strict io;
- a "could return" flag – code after it might not be executed;
- the main `effect_map`;
- a list of levels of executed unpredictable procedures (function parameters);
- a list of pairs of `id` that clash – they could be referring to the same object because of reference collisions.

2.6.2 Merging effects

One of the main parts of the effect analysis is how to merge two effects to produce a resulting effect that combines them. Two effects can be sequential, alternative or commutative, each with an associated merger function.

Let us first examine two "small" effects. For commutative and alternative effects, the resulting effect reads, if either of the input effects reads. For sequential effects, it reads, if the first input effect reads or if the second one reads and the first one does not overwrite.

Figure 1: Merging the states of two effects

(a) Sequential effects

	S	O	P	I
S	S	O	P	I
O	O	O	O	I
P	P	O	P	I
I	I	O	I	I

(b) Alternative effects

	S	O	P	I
S	S	P	P	I
O	P	O	P	I
P	P	P	P	I
I	I	I	I	I

(c) Commutative effects

	S	O	P	I
S	S	O	P	I
O	O	I	I	I
P	P	I	I	I
I	I	I	I	I

Here the letters S, O, P and I represent the states `Same`, `Overwritten`, `Partial` and `Invalid` respectively. The row shows the first effect and the column – the second one.

Deciding what the resulting effect does to the object (as in the state it leaves it in) is a bit more complicated but still quite intuitive. All possibilities are shown in Figure 1. Notice that commutatively merging two effects that both modify the same object leads to it being in an invalid state.

Sequentially merging an effect that invalidates an object with an effect that reads it throws a `Com_error`. Moreover, commutatively merging an effect that modifies an object in any way and one that reads it also throws a `Com_error`. These are the most common types of error.

After we know how to merge single "small" effects we just have to match them up to merge the whole maps. This is done by a higher order function which takes the merger rule as an argument:

```
let combine_ems rule em1 em2 =
  let combine1 id eff1 em =
    let eff2 = match IdMap.find_opt id em2 with
        None -> (false, Same)
      | Some eff -> eff in
    let eff = rule eff1 eff2 in
    IdMap.add id eff em
  and combine2 id eff2 em =
    match IdMap.find_opt id em1 with
        None ->
          let eff = rule (false, Same) eff2 in
          IdMap.add id eff em
      | Some eff -> em in
  let temp = IdMap.fold combine1 em1 IdMap.empty in
  IdMap.fold combine2 em2 temp
```

Merging two `big_effect`s boils down to first handling all of the extra bits and then merging the `effect_map`s. The full codes for the three types of merger will not be shown here because we have not yet explained how to use the other components of the `big_effect`.

2.6.3 Motivation

Here we present our motivation for keeping track of invalid states instead of immediately throwing an error. Our approach offers many benefits in practical and common code patterns. For example, consider the following function which swaps two pairs of values:

```
var x, y, z, w: integer
proc two_swaps();
  var a: integer;
begin
  [ a := x; x := y;  y := a
  : a := z; z := w;  w := a ]
end;
```

While this is perfectly valid, the more naive approach would not see that it is okay to leave `a` in an invalid state. Another common case is when we have some heavy object that we want to pass (by reference) to two functions that will just read some of it. However, we need to modify one/a few of the fields of the object to suite the functions and then restore it after. (The modification is different for the two functions.) With our approach this is perfectly valid even if we do not throw away the object after that, as long as we restore it to a valid state after the commuting statements.

2.6.4 Generating effects

The way to produce `big_effect`s is through several functions and fixed values. The main functions are `read_effect` and `write_effect`. They accept an `id` and its type (given as a `ptype`). Then they `expand` (see below) the `id` to a list of `id`s and generate a `big_effect` that maps all of them to reading/writing respectively. However, `write_effect` needs to consider what sorts of `id`s it has. Regular variables and references get overwritten entirely when we write to them but elements of an array and objects on the heap do not. This is because, we treat all objects on the heap with the same type as the same object (and similarly for elements of an array). Therefore, writing to one of them does not overwrite all objects we would identify with that `id`. Finally, for fields we decide recursively:

```
let write_effect id t =
  let rec state id = match id with
      Var _ -> Overwritten
    | Ref _ -> Overwritten
    | Heap _ -> Partial
    | Element _ -> Partial
    | Field (_, _, id2) -> state id2
    | _ -> failwith "write_effect" in
  let ids = expand id t in
  let effs = List.map (fun id -> (id, (false, state id))) ids
  in (false, false, from_list effs, [], [])
```

We mentioned expanding. This is the process of breaking down a composite object into its building components. Doing that requires knowing its type (not just type id). Here is the recursive `expand` function:

```
let rec expand id t =
  match t.t_guts with
      BasicType _ -> [id]
    | PointerType _ -> [id]
    | ArrayType (_, t2) -> expand (Element (t2.t_id, id)) t2
    | RecordType ds ->
        let expand1 d = expand (Field (d.d_type.t_id, d.d_tag,
             id)) d.d_type in
        List.concat (List.map expand1 ds)
    | _ -> failwith "expand"
```

Clearly, the recursion stops only when it reaches a primitive type or pointer. One further advantage it offers is being able to differentiate between things of the same type on the heap. This is because `picoPascal` does not allow taking a pointer to a value. Therefore, the `x` field of a `Point` on the heap is distinct from its `y` field and also any plain `integer` on the heap. The same (to a slightly smaller extent) goes for telling apart things passed by reference.

2.6.5 Main structure

The main structure of the effect analyzer is the similar to that of the semantic analyzer (even the names of the main functions are the same).

It has a recursive `check_stmt`, which checks a statement for errors and returns its effect. It uses the mutually recursive `check_expr`, which does the same for expressions and `check_id`, which also finds the `id` of the object the expression refers to (or `None`, if it does not refer to an object), as well as `check_funcall`, `check_libcall`, `check_args`, `check_arg` and `check_write`, whose function is also clear. Let us first present several cases of `check_stmt`:

```
    | Commutative (s1, s2) ->
        let eff1 = check_stmt s1
        and eff2 = check_stmt s2 in
        combine_big_com eff1 eff2
```

As can be seen, handling commutative statements does not require any specific code in the skeleton. It just follows the structure of the code while all of the heavy lifting is done by the merging.

```
    | IfStmt (cond, thenpt, elsept) ->
        let cond_eff = check_expr cond
        and then_eff = check_stmt thenpt
        and else_eff = check_stmt elsept in
        let body_eff = combine_big_alt then_eff else_eff in
        combine_big_seq cond_eff body_eff
```

Notice the use of `combine_big_alt` for the `then` and `else` parts – exactly one of them will happen.

```
  | WhileStmt (cond, body) ->
      let cond_eff = check_expr cond
      and body_eff = check_stmt body in
      let maybe_eff = partialize_big body_eff in
      combine_big_seq cond_eff maybe_eff
```

For the `while` and `for` loops we use the helper `partialize_big` function, which turns all overwrites to partial overwrites. This is because the body may not get executed even once. However, this is not the case for `repeat` loops.

```
  | Assign (lhs, rhs) ->
      let (eff1, ido) = check_id lhs
      and eff2 = check_expr rhs in
      let expr_eff = combine_big_seq eff1 eff2 in
      combine_big_seq expr_eff (write_effect (expand (get
          ido) lhs.e_type))
```

Notice that for assignment, writing is done after finding the `id` of the destination and the value (this is equivalent to the order in the code generation). For other simpler assignments (e.g. with library functions for reading) we just use the `check_write`, which sequences the effect for accessing a value and then the one for writing to it. Note: `get` and `exists` are helper functions for dealing with `options`.

The other cases follow a pattern similar to the ones above. The exception is the procedure calling, which delegates its work to the `check_funcall` function.

The `check_expr` function is quite similar. The only thing to point out is how it handles things that could refer to an object. Essentially, it delegates the work to `check_id` and then just applies a `read_effect`. This saves code duplication:

```
 Variable _ | Sub _ | Select _ | Deref _ ->
      let (eff, ido) = check_id e in
      if not (exists ido) then eff
      else combine_big_seq eff (read_effect (get ido) e.
          e_type)
```

And finally let us look at the cases of the `check_id` function:

```
    Variable x ->
      let d = get x.x_def in
      let ido = match d.d_kind with
          VarDef | CParamDef -> Some (Var (e.e_type.t_id, d.
              d_level, d.d_tag))
        | VParamDef -> Some (Ref (e.e_type.t_id, d.d_level,
            d.d_tag))
        | ConstDef _ | StringDef -> None
        | _ -> failwith "check_id" in
      (no_effect, ido)
```

The basic case where we've reached a specific object. The function has to decide its `id` depending on the kind of definition.

```
| Deref e1 ->
    let eff1 = check_expr e1 in
    let ido = Some (Heap e.e_type.t_id) in
    (strictify (eff1), ido)
```

For dereferencing we just return a `Heap id` of the correct type. Notice the use of `strictify`, it makes this a noncommutative effect, if we are strict (or does nothing, if we are not). This is because we might be dereferencing a `null` pointer.

```
| Sub (e1, e2) ->
    let (eff1, ido) = check_id e1
    and eff2 = check_expr e2 in
    let ido2 = match ido with
        None -> None
      | Some id -> Some (Element (e.e_type.t_id, id)) in
    let tot_eff = combine_big_seq eff1 eff2 in
    begin match e2.e_value with
        None -> (strictify tot_eff, ido2)
      | Some n ->
          if n >= 0 && n < bound e1.e_type then (tot_eff,
              ido2)
          else (strictify tot_eff, ido2)
    end
```

Here we just add on an `Element` to the base `id` (if it exists). Also, we need to use `strictify` again because the index might be outside the bounds of the array, which could cause an error. However, we could let it run normally, if we know the index in advance (and it is in bounds). The `Select` case is very similar but it also tags the `id` with the identifier of the field and doesn't use `strictify` (or check the index), this is because everything is known at compile time.

The final trivial case is when we don't have a match – then we delegate to `check_expr` and return a `None`.

After processing the whole statement for some function (or the main program) we check that it leaves all objects it doesn't own in a valid state (this includes its return value and the io). Then we filter its effects to remove its local variables and store the result as the function's effect. This process will be explained in more detail in the section about functions.

What will follow now is a series of sections, each covering a specific language feature. Some will require knowledge from a previous one, but they should be mostly self contained.

2.6.6 Returns

Handling returns consists of two parts: the return value itself and the fact that the function stops execution. The second part is handled by one of the flags of `big_effect`. When combining two effects sequentially, if the first one can return, we `partialize` all writes from the second one. When combining them commutatively, it might be tempting to just throw an error, but what we should actually do is invalidate all of the writes of the other one (the one that doesn't return). This is because they might not happen. The logical conclusion is that, if both sides can return, both should be invalidated. Actually, we oversimplified a bit – when we partialize and/or invalidate, we only do it for objects the current function doesn't own. This is because either the function returns – then the state of its local variables doesn't matter, or it doesn't return – then we don't need to modify any of the local states.

On the other hand, returning a value is treated as writing to a Return id (as well as setting the return flag). This prevents us from returning different values in two commutative statements, as that would invalidate the Return object.

These two possibilities are captured by the return_void_effect and the return_value_effect. Then the decision which to use is handled by the following case of the check_stmt function:

```
| Return res ->
    if not (exists res) then return_void_effect
    else combine_big_seq (check_expr (get res))
        return_value_effect
```

2.6.7 References – the inside view

Any reference (var parameter) can be viewed from two points: the one passing it and the one getting it. Here, we focus on the second point of view.

The main difficulty with references comes from the fact that they could be referring to almost any other object of the same type. They can't be referring to a non-reference object that is local to the scope of the function where the reference was passed as an argument. This is quite intuitive, as these objects did not exist at the time the reference was created. So, to check whether two ids can be referring to the same object, we just need to check whether at least one is a reference and whether the levels are right. However, because of this expanding that we do (breaking compound objects apart) we may actually need to trace both ids to their roots, checking that they are a match at each step. This is done by the ref_match function. Its implementation is obvious once we have the idea, so it will not be shown here.

Now on every commutative merger we can find all possible reference clashes between the two statements. Then, if one of them reads the object and the other writes to it, we throw an error. However, if both just write we need to invalidate something. We can just invalidate both but that is not optimal. There are two cases: they do not actually refer to the same object, then everything is fine; they do refer to the same object, then writing to either of them later will validate the object again. So, what we actually do is register a conflict between these two ids. Then in sequential mergers, if the second effect reads from an id that is in a conflict in the first effect, we throw an error, however if it overwrites an id in a conflict, we remove the conflict.

Finally, when we finish processing a function's statement we also check that its effect has no reference conflicts.

2.6.8 Functions

In order to properly handle functions we need to process them in an order such that each function is processed after every function that it calls (ignoring recursion). So, we build a directed graph where the nodes are functions and the edges are function calls and we process its strongly connected components in reverse topological order. The graph will be represented as an array of `nodes`:

```
type node =
  {  n_index: int;
     n_body: stmt;
     n_level: int;
     n_params: def list;
     mutable n_calls: int list;
     mutable n_called: int list;
     mutable n_effect: big_effect option;
     mutable n_argeffs: effect_map option list;
     mutable n_visit: bool;
     mutable n_scc: int }
```

Each node stores: its index, its statement, its level, the definitions of its parameters, the indices of the nodes it calls, the indices of the nodes that call it, its external effect, its effect on its `var` parameters and finally two utility values.

Building the graph consists of two stages. First, we count the number of functions, so we can create an array of that size. Then, we pass through the whole code recursively and we fill the array by calling `make_node` and add edges by calling `add_dependency`. Also, we mark each function definition with its node's index using the newly added field `d_node`.

After the graph is built, we use Kosaraju's algorithm on the transpose graph to get the strongly connected components in reverse topological order:

```
let kosaraju_check num_nodes =
  let stack = ref [] in
  let rec visit x =
    let n = !nodes.(x) in
    if not n.n_visit then begin
      n.n_visit <- true;
      List.iter visit n.n_called;
      stack := x :: !stack
    end in
  for x = 0 to num_nodes - 1 do visit x done;
  let rec assign root x =
    let n = !nodes.(x) in
    if n.n_scc = - 1 then begin
      n.n_scc <- root;
      x :: List.concat (List.map (assign root) n.n_calls);
    end else [] in
  let handle x =
    if !nodes.(x).n_scc = -1 then check_scc (assign x x) in
  List.iter handle !stack
```

This is a standard algorithm. The basic idea is to sort the nodes by decreasing finishing time in DFS and then run a second DFS on the transpose graph in that order. Here, we do the first DFS on the transpose and the second – on the regular one in order to get the SCCs in the order we need. Then for each (non-empty) SCC we call `check_scc`.

If the component only contains a single node and no loop, it is non-recursive, so we just call the `check_node` function once. Otherwise, we need to do two passes. First, we give all nodes a `no_effect`, then we find their effects under that assumption. After that, we combine all of their effects with an alternative merger (so essentially taking the union) and assign each node this as its effect. It is quite obvious that each node's effect is now strictly stronger than its actual effect (or precisely equal). Now, we can just run `check_node` on each of them. Note that we disable errors during the preliminary pass.

However, testing showed that this approach detected a lot of correct programs as incorrect. This was fixed by actually having two preliminary passes. First, we proceed as normal but after the first pass, instead of merging the effects, we just "clean" them up. By this we mean that we remove any potentially undesirable effects that may not actually happen. In practice, this means leaving only overwrites (removing reads, partials and invalids). Then, we do a regular pass with no "cleaning" and we merge the effects and do the final pass. This is correct (as in it doesn't cause false positives) because we are not introducing any non-existent effects and we still have a full pass at the end.

The `check_node` function simply runs `check_stmt` and verifies that every non-local object is left in a valid state. Finally, it stores the external effects in the node's `n_effect` field (and deals with `n_argeffs`). The second two parts are done by `keep_above`:

```
let keep_above lvl (nc, rt, em, up, cf) =
  let rec keep_one id =
    match id with
        Var (_, lvl2, _) -> (lvl2 < lvl, lvl2 < lvl)
      | Ref (_, lvl2, _) -> (true, lvl2 < lvl)
      | Heap _ -> (lvl > 0, lvl > 0)
      | Element (_, id2) -> keep_one id2
      | Field (_, _, id2) -> keep_one id2
      | Return -> (true, false)
      | IO -> (true, true) in
  let one_iter id (_, s) =
    let (keep_valid, keep) = keep_one id in
    if invalid s && keep_valid then com_error "Invalid state
        at return\n"; keep
  and keep_unpred lv = lv < lvl in
  if cf <> [] then com_error "Reference conflict at return\n";
  (nc, false, IdMap.filter one_iter em, List.filter
      keep_unpred up, [])
```

You may notice the `keep_unpred` part, it is used for handling function parameters, so it will be explained in a later section.

This explains how we find the effects of functions. Using these effects is much simpler. As already mentioned, finding the effects of a function/procedure call is done by `check_funcall`. It has a two cases. The first one is for library functions, then it just delegates to `check_libcall`. The second one deals with both regular functions and function parameters.

For library functions, we just pattern match on the function and we have the effects of each one hard-coded. The only interesting one is for binary operators: in the case of division and modulo, if we are being strict, and we don't know that we are dividing by a non-zero number, we have to use `strictify`. The same goes for a few other library functions such as allocating memory.

Library functions are also where we use the `io_effect`. It consists of partially writing to the `IO id`. This is because we are not overwriting the io, just "appending" to it (no matter whether we are inputting or outputting). If we are being strict, we use a `noncommutative_effect` instead.

For regular functions we combine the arguments' effects with the external effect of the function. For `CParamDef` parameters we use the effect for evaluating the expression. How we handle other cases will be explained after covering all other relevant parts.

2.6.9 References – the outside view

After we have the infrastructure for properly processing functions, storing each function's effects on each of its arguments will not be too difficult. Then we can use these stored effects instead of assuming that `var` parameters are always read from and written to.

First, in `check_node`, after verifying everything and extracting the external effect, we similarly extract the effects on the arguments. For each argument we just store an `effect_map` instead of a full `big_effect`. This is mostly for convenience, as the extra flags and data are not relevant. Actually, the function `check_node` accepts an extra flag – whether we want it to update the `argeffs`. This is because we shouldn't do that in the preliminary passes for recursive functions. (Upon initialization nodes assume that they read from and partially write to all of their reference arguments.) Here is the completed `check_node`:

```
let check_node update_argeffs n =
  level := n.n_level;
  let eff = check_stmt n.n_body in
  n.n_effect <- Some (keep_above n.n_level eff);
  if update_argeffs then
    n.n_argeffs <- get_argeffs n.n_params eff
```

Note that each parameter may have more that one "small" effect associated with it. This is because it might be a compound type. In that case each relevant `id` has the same `Ref` as a base and then branches from that.

Now comes the next part – how to use the stored `argeffs`. The challenge here is that, when we are calling a function, we need to match up the following three things: the definitions of the formals, the argument effects and the actual arguments we pass. This is handled by the `check_args` function:

```
and check_args formals argeffs args =
  let rec map3 f xs ys zs = match xs, ys, zs with
      x::xs, y::ys, z::zs -> f x y z :: map3 f xs ys zs
    | _ -> [] in
  let cmb (r_eff1, w_eff1) (r_eff2, w_eff2) =
    let r_eff3 = combine_big_seq r_eff1 r_eff2
    and w_eff3 = combine_big_seq w_eff1 w_eff2 in
    (r_eff3, w_eff3) in
  let rw_effs = map3 check_arg formals argeffs args in
  List.fold_left cmb (no_effect, no_effect) rw_effs
```

Notice that we are actually keeping track of two `big_effects` – one for reading and one for writing. This is because in `check_funcall` we need to apply all of the reading effects (the ones for passing the arguments) before the external effects of the function but we need to apply all of its writing effects (what the function does to its arguments) after its external effects. This is equivalent to applying them at the points where the function actually did the writing because of the guarantee that everything is left in a valid state.

These are the two relevant cases of `check_arg`:

```
      CParamDef -> (check_expr arg, no_effect)
    | VParamDef ->
        let (eff, ido) = check_id arg in
        let (read_eff, write_eff) = reroot (get ido) formal.
            d_type argeff in
        (combine_big_seq read_eff eff, write_eff)
```

For pass-by-value parameters, the only effect is for evaluating the expression. However, for pass-by-reference parameters, the additional read and write effects are found by rerooting the `id`s in the `argeff` to have the current `id` as their base. In some cases we pass `None` as the `argeff` (which means we do not know what happens to the argument, so we assume the worst); the rerooting function needs to know the type in that case, so it can expand the `id` appropriately.

```
let reroot id t emo =
  let rec reroot_one arg_id = match arg_id with
      Ref _ -> id
    | Element (t, id2) -> Element (t, reroot_one id2)
    | Field (t, x, id2) -> Field (t, x, reroot_one id2)
    | Var _ | Heap _ | Return | IO -> failwith "reroot_one" in
  let accum arg_id (r, s) (rem, wem) =
    let new_id = reroot_one arg_id in
    let new_rem = if r then
      IdMap.add new_id (r, Same) rem else rem
    and new_wem = if changed s then
      IdMap.add new_id (false, s) wem else wem in
    (new_rem, new_wem) in
  if exists emo then
    let (rem, wem) = IdMap.fold accum (get emo) (IdMap.empty,
        IdMap.empty) in (em_effect rem, em_effect wem)
  else (read_effect id t, partial_effect id t)
```

2.6.10 Function parameters

Function parameters are the last language feature we have not discussed. However, their handling is actually quite easy. Each time we call a function parameter we just assume its effect is unpredictable (as are its effects on its `var` parameters, so we just pass a list of `None`s for them). We can discard that assumption once we know what was actually passed, when we are analysing the caller. Because of that, each `unpredictable_effect` is tagged with the level where it was passed as an argument; when we are done with that level (so at the `keep_above` part), we remove it (we remove all unpredictable effects with level no less than the current one).

Unpredictable effects mostly matter when we are merging effects. We assume they read and may write to anything they could have access to. This is similar to how we check for reference collisions. Any object with a level not less than that of the function parameter could not be accessed by it, so it is safe to write to it in the other half of a commutative statement. However, if we find that there is a clash, we throw an error. When sequentially combining effects, if the second one has unpredictable components, we demand that everything they can access is left in a valid state by the first effect. Note: everything has access to the io, the heap and any object referred to by a reference. Because of this (the io part), if we are being strict, we make `unpredictable_effect`s noncommutative.

The final part left is how to handle the function parameter from the caller's side. By that point, we have removed the unpredictable component. What we do instead is assume that the one passing the function is calling it (though `partialize`d because we aren't sure it will be called). If the caller is passing a regular function, he already knows its effects, and if it is a function parameter, the whole process repeats again with him adding a new `unpredictable_effect`. This is the relevant case of `check_arg`:

```
| PParamDef ->
    let d = match arg.e_guts with
        Variable x -> get x.x_def
      | _ -> failwith "check_arg" in
    begin match d.d_kind with
        ProcDef -> (partialize_big (get !nodes.(d.d_node).
            n_effect), no_effect)
      | PParamDef -> (unpredictable_effect d.d_level,
          no_effect)
      | _ -> failwith "check_arg"
    end
```

Because we treat passing function parameters as calling them, we also add them as a dependency in the graph building stage but this is a trivial addition.

We have now covered how everything related to functions is handled – returns, references, function parameters and the order in which we analyse the functions. This concludes our implementation, as all language features are now supported quite well.

2.7 Testing

To facilitate testing, changes were made to the the `Makefile`. First, we needed to add the files for compilation. Also, the different nature of the task required us to abandon the previous method of testing. Instead, now all tests are prefixed by either `right_` or `wrong_`. The `test` command verifies that all `wrong_` tests are detected as "Possibly incorrect". Then, it goes through the `right_` tests but there detecting them as "Correct" is not necessary (misclassifications are just printed on the console). There is also the `teststrict` command, which does the same but with the `-strictio` flag enabled.

We have added 98 new tests (most of which are quite small). The high number is so we can thoroughly test all possible fail conditions and also near-fail conditions (that are actually correct). Here we will present parts of some of them.

A small correct function that illustrates the basic functionality:

```
proc next_two(x: integer);
  var y, z: integer;
begin
  [ y := x + 1 : z := x + 2 ];
  print_num(y); print_num(z)
end;
```

The two swaps function that we already mentioned. It illustrates the point of invalid states:

```
var x, y, z, w: integer
proc two_swaps();
  var a: integer;
begin
  [ a := x; x := y;  y := a
  : a := z; z := w;  w := a ]
end;
```

An illustration of a wrong program. It demonstrates heap clashes:

```
var p, q: pointer to integer;
var x: integer;
begin
  new(p);  q := p;
  [ x := p^ : q^ := 7 ];
  print_num(x); newline()
end.
```

In this correct program we can see that different types do not clash:

```
var p: pointer to integer;
var q: pointer to char;
begin
  new(p); new(q);
  [ p^ := 5 : q^ := 'd' ];
  print_num(p^); print_char(q^)
end.
```

Here we can see how different fields of a record do not clash:

```
type Point = record x, y: integer end;
proc init(var p: Point);
begin
  [ p.x := 0 : p.y := 2 ]
end;
```

And for completeness, let us show an example of an array index clash:

```
var a: array 10 of integer;
proc setij(i, j: integer);
begin
  [ a[i] := 5 : a[j] := 2 ]
end;
```

Now that we have shown the basic functionality, let us demonstrate some more interesting examples. This programs show an invalid object becoming valid again. The example doesn't really show a realistic reason to have the invalid state. This is done in order to keep it short. Also, notice that `x` is modified in both branches, which makes this correct:

```
var x: integer;
var y: integer;
begin
  [ x := 1 : x := 2 ];
  if y = 5 then x := 2
  else [ x := 4 : y := 2 ] end;
  print_num(x); newline();
  print_num(y); newline()
end.
```

Similarly, `case` statements work like `if` statements with more branches, so we will not include an example here.

This wrong function shows that the body of a `while` loop may not get executed. It also demonstrates the need to keep the heap valid:

```
type ptr = pointer to integer;
proc test(p: ptr; n: integer);
begin
  [ p^ := 5 : p^ := 2 ];
  while n < 100 do
    p^ := n;
    n := n * n
  end
end;
```

Again, we will not include an example of a `for` loop, as the idea there is quite similar.

However, a similar function with a `repeat` loop is correct because it will get executed at least once:

```
proc test(var a: integer; n: integer);
begin
  [ a := 12 : a := -5 ];
  repeat
    a := n; n := n * n
  until n >= 100
end;
```

Now, that we have covered the basic control flow structures, we should look at functions. This correct function demonstrates having `void return`s on both sides of a commutative statement:

```
proc f(x, y: integer);
begin
  [ if x <= 0 then return end
  : if y <= 0 then return end ];
  print_num(x + y); newline()
end;
```

And this one demonstrates writing to local variables on the other side:

```
proc positive_cube(x: integer): boolean;
  var res: integer;
begin
  [ if x <= 0 then return false end
  : res := x * x * x ];
  print_num(res); newline();
  return true
end;
```

However, this function is incorrect, as both sides can return different results:

```
proc f(x: integer): integer;
begin
  [ return x : if x < 0 then return 7 end ]
end;
```

Let us now show a couple of interesting examples with reference collisions. This is the basic case of a potential reference collision and so it is incorrect:

```
var x: integer;
proc f(var y: integer);
begin
  [ x := 1 : y := 3 ]
end;
```

And here we have a read-write reference collision:

```
var x: integer;
proc f(var y: integer);
  var a: integer;
begin
  [ a := x : y := 3 ];
  print_num(a); newline()
end;
```

This correct example demonstrates that variables local to the function that got the reference cannot collide with it:

```
proc f(var y: integer);
  proc g();
    var a: integer;
    proc h();
    begin
      [ a := 2 : y := 8 ]
    end;
  begin
    h(); print_num(a)
  end;
begin
  g(); print_num(y)
end;
```

And here we have a more convoluted reference collision between an integer and an integer field of a point:

```
type point = record x, y: integer end;
var p: point;
proc f(var x: integer);
begin
  [ p.x := 5 : x := 3 ];
  print_num(x); newline()
end;
```

Now let us present a few examples, where knowing the effects of functions is important. This is the most basic case of correct use:

```
var x, y: integer;
proc f();
begin
  print_num(x); newline()
end;
proc g();
begin
  [ y := x : f() ];
  print_num(y); newline()
end;
```

And here is another correct example that shows the need to know how each function treats its arguments:

```
type Point = record x, y: integer end;

proc print_x(var p: Point);
begin print_num(p.x); newline() end;

proc increment_y(var p: Point; x: integer);
begin p.y := p.y + x end;
```

```
proc weird(var p: Point): integer;
begin
  [ print_x(point) : increment_y(point, 5) ];
  return p.x + p.y
end;
```

And a correct example with recursion. What better than Fibonacci numbers:

```
proc fib(n: integer): integer;
  var x, y: integer;
begin
  case n of
      0: return 0
    | 1: return 1
  else
    [ x := fib(n-1) : y := fib(n-2) ];
    return x + y
  end
end;
```

And also this very weird but correct mutually recursive setup. Notice that, without the call to `bar`, `foo` would be incorrect because one side is reading `x` and the other is writing to it. Before the addition of the third pass, this example only worked if we ordered the two functions opposite of how they are now. However, after the improvements we can deal with it (and more complicated examples) no matter how we arrange the functions:

```
var x, y: integer;
proc foo(z: integer);
begin
  [ bar(z); y := x : x := 2 ];
  print_num(y); newline();
  x := 0
end;
proc bar(z: integer);
begin
  if z <= 0 then x := 5; return
  else foo(z - 1); x := z end
end;
```

Now that we have given some correct uses, let us see a few incorrect ones. This one wrong because both sides can print:

```
proc foo(x: integer);
  proc bar(y: integer);
  begin foo(x + y) end;

begin
  if x > 100 then return end;
  [ foo(x + 1) : bar(x) ];
  print_num(x); newline()
end;
```

And this one is incorrect for obvious reasons. It is another example of knowing what a function does with its arguments:

```
type Point = record x, y: integer end;
proc print_sum(var p: Point);
begin
  print_num(p.y + p.x); newline()
end;
proc sum_reset(var p: Point);
begin
  [ print_sum(p) : p.x := 0 ]
end;
```

Finally, a few examples with function parameters. This one is correct because no matter what `h` is, it does not have access to `a`. Note that we could not commute the printing with the call to `h`, as `h` might also affect the io. However this use is okay:

```
proc f(proc h(y: integer));
  var a: integer;
begin
  [ a := 2 : h(8) ];
  print_num(a); newline()
end;
```

This is wrong, because it is possible that `use_func` uses the procedure it was given (as will indeed happen). In that case we would get a problem with the order of the io:

```
proc use_func(proc h(y: integer));
begin
  h(5)
end;
proc print_x(x: integer);
begin
  print_num(x); newline()
end;
begin
  [ print_string("Hello!\n") : use_func(print_x) ]
end.
```

A lot more tests can be found in the diff but we feel that these demonstrate the functionality of all language features as well as a few tricky cases that naive solutions will not accept.

For interest, out of the 46 correct tests 14 fail under the stricter conditions. However, in practice with more complicated functions that percentage will probably increase as there will be more places where errors could occur.

```
--- a/check.ml
+++ b/check.ml
@@ -90,7 +90,7 @@ let try_binop w v1 v2 =
 (* |has_value| -- check if object is suitable for use in expressions *)
 let has_value d =
   match d.d_kind with
-      ConstDef _ | VarDef | CParamDef | VParamDef | StringDef -> true
+      ConstDef _ | VarDef | RefDef | CParamDef | VParamDef | StringDef -> true
     | _ -> false

 (* |check_var| -- check that expression denotes a variable *)
@@ -100,7 +100,7 @@ let rec check_var e addressible =
         let d = get_def x in
         begin
           match d.d_kind with
-              VarDef | VParamDef | CParamDef ->
+              VarDef | RefDef | VParamDef | CParamDef ->
                 d.d_mem <- d.d_mem || addressible
             | _ ->
                 sem_error "$ is not a variable" [fId x.x_name]
@@ -362,7 +362,22 @@ let rec check_stmt s env alloc =
            be used to save the upper bound.  *)
        let d = make_def (intern "*upb*") VarDef integer in
        alloc d; upb := Some d
-
+
+    | RangeStmt (x, arr, body, iter_ref, upb_ref) ->
+        let at = check_expr arr env in
+        begin
+          match at.t_guts with
+              ArrayType (_, bt) ->
+                let iter = make_def x RefDef bt in
+                alloc iter; iter_ref := Some iter;
+                let upb = make_def (intern "*iter_upb*") RefDef bt in
+                alloc upb; upb_ref := Some upb;
+                let env' = add_def iter env in
+                check_stmt body env' alloc
+            | _ ->
+                sem_error "range-based for loop over non-array" []
+        end
+
     | CaseStmt (sel, arms, deflt) ->
         let st = check_expr sel env in
         if not (scalar st) then
@@ -401,14 +416,18 @@ let upward_alloc size d =
   size := !size + r.r_size;
   d.d_addr <- Local addr

+let is_reference d = match d.d_kind with
+    RefDef -> true
+  | _ -> false
+
 (* local_alloc -- allocate locals downward in memory *)
 let local_alloc size nreg d =
-  if !regvars && not d.d_mem && scalar (d.d_type)
+  if !regvars && not d.d_mem && (scalar d.d_type || is_reference d)
       && !nreg < Mach.nregvars then begin
     d.d_addr <- Register !nreg; incr nreg
   end
   else begin
-    let r = d.d_type.t_rep in
+    let r = if is_reference d then addrtype.t_rep else d.d_type.t_rep in
     align r.r_align size;
     size := !size + r.r_size;
     d.d_addr <- Local (local_base - !size)
```

```
@@ -434,7 +453,7 @@ let global_alloc d =
 let do_alloc alloc ds =
   let h d =
     match d.d_kind with
-        VarDef | CParamDef | VParamDef | FieldDef | PParamDef ->
+        VarDef | RefDef | CParamDef | VParamDef | FieldDef | PParamDef ->
           alloc d
       | _ -> () in
   List.iter h ds
--- a/dict.ml
+++ b/dict.ml
@@ -78,6 +78,7 @@ type def_kind =
   | StringDef                   (* String *)
   | TypeDef                     (* Type *)
   | VarDef                      (* Variable *)
+  | RefDef                      (* Reference *)
   | CParamDef                   (* Value parameter *)
   | VParamDef                   (* Var parameter *)
   | FieldDef                    (* Field of record *)
--- a/dict.mli
+++ b/dict.mli
@@ -41,6 +41,7 @@ type def_kind =
   | StringDef                   (* String constant *)
   | TypeDef                     (* Type *)
   | VarDef                      (* Variable *)
+  | RefDef                      (* Reference *)
   | CParamDef                   (* Value parameter *)
   | VParamDef                   (* Var parameter *)
   | FieldDef                    (* Field of record *)
--- a/lexer.mll
+++ b/lexer.mll
@@ -18,7 +18,7 @@ let symtable =
       ("end", END); ("of", OF); ("proc", PROC); ("record", RECORD);
      ("return", RETURN); ("then", THEN); ("to", TO);
     ("type", TYPE); ("var", VAR); ("while", WHILE);
-      ("pointer", POINTER); ("nil", NIL);
+      ("pointer", POINTER); ("nil", NIL); ("in", IN);
      ("repeat", REPEAT); ("until", UNTIL); ("for", FOR);
     ("elsif", ELSIF); ("case", CASE);
      ("and", MULOP And); ("div", MULOP Div); ("or", ADDOP Or);
--- a/parser.mly
+++ b/parser.mly
@@ -21,7 +21,7 @@ open Tree
 /* keywords */
 %token                  ARRAY BEGIN CONST DO ELSE END IF OF
 %token                  PROC RECORD RETURN THEN TO TYPE
-%token                  VAR WHILE NOT POINTER NIL
+%token                  VAR WHILE NOT POINTER NIL IN
 %token                  REPEAT UNTIL FOR ELSIF CASE

 %type <Tree.program>    program
@@ -119,6 +119,7 @@ stmt1 :
   | FOR name ASSIGN expr TO expr DO stmts END
                                        { let v = makeExpr (Variable $2) in
                                          ForStmt (v, $4, $6, $8, ref None) }
+  | FOR IDENT IN variable DO stmts END  { RangeStmt ($2, $4, $6, ref None, ref None) }
  | CASE expr OF arms else_part END     { CaseStmt ($2, $4, $5) } ;

 elses :
--- a/tgen.ml
+++ b/tgen.ml
@@ -92,7 +92,7 @@ let rec gen_addr v =
          match d.d_kind with
              VarDef ->
               address d
```

```
-             | VParamDef ->
+             | RefDef | VParamDef ->
                  <LOADW, address d>
              | CParamDef ->
                  if scalar d.d_type || is_pointer d.d_type then
@@ -321,6 +321,24 @@ let rec gen_stmt s =
              <JUMP l1>,
              <LABEL l2>>

+       | RangeStmt (_, arr, body, iter_ref, upb_ref) ->
+           let iter = match !iter_ref with Some d -> d | _ -> failwith "range for"
+           and upb = match !upb_ref with Some d -> d | _ -> failwith "range for" in
+           let l1 = label () and l2 = label () in
+           <SEQ,
+             <STOREW, gen_addr arr, address iter>,
+             <STOREW,
+               <OFFSET, <LOADW, address iter>, <CONST arr.e_type.t_rep.r_size>>,
+               address upb>,
+             <JUMP l2>,
+             <LABEL l1>,
+             gen_stmt body,
+             <STOREW,
+               <OFFSET, <LOADW, address iter>, <CONST (base_type arr.e_type).t_rep.r_size>>,
+               address iter>,
+             <LABEL l2>,
+             <JUMPC (Neq, l1), <LOADW, address iter>, <LOADW, address upb>>>
+
       | CaseStmt (sel, arms, deflt) ->
           (* Use one jump table, and hope it is reasonably compact *)
           let deflab = label () and donelab = label () in
--- a/tree.ml
+++ b/tree.ml
@@ -38,6 +38,7 @@ and stmt_guts =
    | WhileStmt of expr * stmt
    | RepeatStmt of stmt * expr
    | ForStmt of expr * expr * expr * stmt * def option ref
+   | RangeStmt of ident * expr * stmt * def option ref * def option ref
    | CaseStmt of expr * (expr * stmt) list * stmt

 and expr =
@@ -147,6 +148,8 @@ and fStmt s =
         fMeta "(REPEAT $ $)" [fStmt body; fExpr test]
     | ForStmt (var, lo, hi, body, _) ->
         fMeta "(FOR $ $ $ $)" [fExpr var; fExpr lo; fExpr hi; fStmt body]
+    | RangeStmt (x, arr, body, _, _) ->
+        fMeta "(RANGE_FOR $ $ $)" [fId x; fExpr arr; fStmt body]
     | CaseStmt (sel, arms, deflt) ->
         let fArm (lab, body) = fMeta "($ $)" [fExpr lab; fStmt body] in
         fMeta "(CASE $ $ $)" [fExpr sel; fList(fArm) arms; fStmt deflt]
--- a/tree.mli
+++ b/tree.mli
@@ -53,6 +53,7 @@ and stmt_guts =
    | WhileStmt of expr * stmt
    | RepeatStmt of stmt * expr
    | ForStmt of expr * expr * expr * stmt * def option ref
+   | RangeStmt of ident * expr * stmt * def option ref * def option ref
    | CaseStmt of expr * (expr * stmt) list * stmt

 and expr =
--- /dev/null
+++ b/test/range.p
@@ -0,0 +1,83 @@
+var arr: array 10 of integer;
+var i: integer;
+
+begin
+  i := 0;
+  for x in arr do
+    x := i;
+    i := i + 1
+  end;
+  for x in arr do
+    print_num(x);
+    newline()
+  end
+end.
+
+(*<<
+0
+1
+2
+3
+4
+5
+6
+7
+8
+9
+>>*)
+
+(*[[
+@ picoPascal compiler output
+	.include "fixup.s"
+	.global pmain
+
+	.text
+pmain:
+	mov ip, sp
+	stmfd sp!, {r4-r10, fp, ip, lr}
+	mov fp, sp
+	sub sp, sp, #8
+@   i := 0;
+	mov r0, #0
+	set r1, _i
+	str r0, [r1]
+@   for x in arr do
+	set r4, _arr
+	add r5, r4, #40
+	b .L3
+.L2:
+@     x := i;
+	set r7, _i
+	ldr r0, [r7]
+	str r0, [r4]
+@     i := i + 1
+	ldr r0, [r7]
+	add r0, r0, #1
+	str r0, [r7]
+	add r4, r4, #4
+.L3:
+	cmp r4, r5
+	bne .L2
+@   for x in arr do
+	set r6, _arr
```

```
+       add r0, r6, #40
+       str r0, [fp, #-4]
+       b .L5
+.L4:
+@     print_num(x);
+       ldr r0, [r6]
+       bl print_num
+@     newline()
+       bl newline
+       add r6, r6, #4
+.L5:
+       ldr r0, [fp, #-4]
+       cmp r6, r0
+       bne .L4
+       ldmfd fp, {r4-r10, fp, sp, pc}
+       .ltorg
+
+       .comm _arr, 40, 4
+       .comm _i, 4, 4
+@ End
+]]*)
--- /dev/null
+++ b/test/range_for.p
@@ -0,0 +1,87 @@
+var arr: array 10 of integer;
+var i: integer;
+
+begin
+  for i := 0 to 9 do
+    arr[i] := i;
+  end;
+  for i := 0 to 9 do
+    print_num(arr[i]);
+    newline()
+  end
+end.
+
+(*<<
+0
+1
+2
+3
+4
+5
+6
+7
+8
+9
+>>*)
+
+(*[[
+@ picoPascal compiler output
+       .include "fixup.s"
+       .global pmain
+
+       .text
+pmain:
+       mov ip, sp
+       stmfd sp!, {r4-r10, fp, ip, lr}
+       mov fp, sp
+@   for i := 0 to 9 do
+       mov r0, #0
+       set r1, _i
+       str r0, [r1]
+       mov r4, #9
```

```
+.L2:
+       set r6, _i
+       ldr r7, [r6]
+       cmp r7, r4
+       bgt .L3
+@      arr[i] := i;
+       set r0, _arr
+       lsl r1, r7, #2
+       add r0, r0, r1
+       str r7, [r0]
+@    end;
+       ldr r0, [r6]
+       add r0, r0, #1
+       str r0, [r6]
+       b .L2
+.L3:
+@    for i := 0 to 9 do
+       mov r0, #0
+       set r1, _i
+       str r0, [r1]
+       mov r5, #9
+.L4:
+       set r6, _i
+       ldr r7, [r6]
+       cmp r7, r5
+       bgt .L1
+@      print_num(arr[i]);
+       set r0, _arr
+       lsl r1, r7, #2
+       add r0, r0, r1
+       ldr r0, [r0]
+       bl print_num
+@      newline()
+       bl newline
+       ldr r0, [r6]
+       add r0, r0, #1
+       str r0, [r6]
+       b .L4
+.L1:
+       ldmfd fp, {r4-r10, fp, sp, pc}
+       .ltorg
+
+       .comm _arr, 40, 4
+       .comm _i, 4, 4
+@ End
+]]*)
--- /dev/null
+++ b/test/range_multidim.p
@@ -0,0 +1,136 @@
+var a: array 10 of array 10 of integer;
+var i, cnt: integer;
+
+begin
+  for b in a do
+    cnt := i;
+    for x in b do
+      x := cnt;
+      cnt := cnt + 1
+    end;
+    i := i + 1
+  end;
+  for b in a do
+    cnt := i;
+    for x in b do
+      print_num(x);
+      print_char(' ')
+    end;
+    newline()
+  end
+end.
+
+(*<<
+0 1 2 3 4 5 6 7 8 9
+1 2 3 4 5 6 7 8 9 10
+2 3 4 5 6 7 8 9 10 11
+3 4 5 6 7 8 9 10 11 12
+4 5 6 7 8 9 10 11 12 13
+5 6 7 8 9 10 11 12 13 14
+6 7 8 9 10 11 12 13 14 15
+7 8 9 10 11 12 13 14 15 16
+8 9 10 11 12 13 14 15 16 17
+9 10 11 12 13 14 15 16 17 18
+>>*)
+
+(*[[
+@ picoPascal compiler output
+       .include "fixup.s"
+       .global pmain
+
+       .text
+pmain:
+       mov ip, sp
+       stmfd sp!, {r4-r10, fp, ip, lr}
+       mov fp, sp
+       sub sp, sp, #24
+@   for b in a do
+       set r4, _a
+       add r5, r4, #400
+       b .L3
+.L2:
+@     cnt := i;
+       set r0, _i
+       ldr r0, [r0]
+       set r1, _cnt
+       str r0, [r1]
+@     for x in b do
+       mov r6, r4
+       add r0, r6, #40
+       str r0, [fp, #-4]
+       b .L5
+.L4:
+@       x := cnt;
+       set r7, _cnt
+       ldr r0, [r7]
+       str r0, [r6]
+@       cnt := cnt + 1
+       ldr r0, [r7]
+       add r0, r0, #1
+       str r0, [r7]
+       add r6, r6, #4
+.L5:
+       ldr r0, [fp, #-4]
+       cmp r6, r0
+       bne .L4
+@     i := i + 1
+       set r7, _i
+       ldr r0, [r7]
+       add r0, r0, #1
+       str r0, [r7]
+       add r4, r4, #40
+.L3:
+       cmp r4, r5
+       bne .L2
+@   for b in a do
+       set r7, _a
+       str r7, [fp, #-8]
+       add r0, r7, #400
+       str r0, [fp, #-12]
+       b .L7
+.L6:
+@     cnt := i;
+       set r0, _i
+       ldr r0, [r0]
+       set r1, _cnt
+       str r0, [r1]
+@     for x in b do
+       ldr r7, [fp, #-8]
+       str r7, [fp, #-16]
+       add r0, r7, #40
+       str r0, [fp, #-20]
+       b .L9
+.L8:
+@       print_num(x);
+       ldr r0, [fp, #-16]
+       ldr r0, [r0]
+       bl print_num
+@       print_char(' ')
+       mov r0, #32
+       bl print_char
+       ldr r0, [fp, #-16]
+       add r0, r0, #4
+       str r0, [fp, #-16]
+.L9:
+       ldr r0, [fp, #-16]
+       ldr r1, [fp, #-20]
+       cmp r0, r1
+       bne .L8
+@     newline()
+       bl newline
+       ldr r0, [fp, #-8]
+       add r0, r0, #40
+       str r0, [fp, #-8]
+.L7:
+       ldr r0, [fp, #-8]
+       ldr r1, [fp, #-12]
+       cmp r0, r1
+       bne .L6
+       ldmfd fp, {r4-r10, fp, sp, pc}
+       .ltorg
+
+       .comm _a, 400, 4
+       .comm _i, 4, 4
+       .comm _cnt, 4, 4
+@ End
+]]*)
--- /dev/null
+++ b/test/range_multidim_for.p
@@ -0,0 +1,142 @@
+var a: array 10 of array 10 of integer;
+var i, j: integer;
+
+begin
+  for i := 0 to 9 do
+    for j := 0 to 9 do
+      a[i][j] := i + j
+    end;
+  end;
+  for i := 0 to 9 do
+    for j := 0 to 9 do
+      print_num(a[i][j]);
+      print_char(' ')
+    end;
+    newline()
+  end
+end.
+
+(*<<
+0 1 2 3 4 5 6 7 8 9
+1 2 3 4 5 6 7 8 9 10
+2 3 4 5 6 7 8 9 10 11
+3 4 5 6 7 8 9 10 11 12
+4 5 6 7 8 9 10 11 12 13
+5 6 7 8 9 10 11 12 13 14
+6 7 8 9 10 11 12 13 14 15
+7 8 9 10 11 12 13 14 15 16
+8 9 10 11 12 13 14 15 16 17
+9 10 11 12 13 14 15 16 17 18
+>>*)
+
+(*[[
+@ picoPascal compiler output
+       .include "fixup.s"
+       .global pmain
+
+       .text
+pmain:
+       mov ip, sp
+       stmfd sp!, {r4-r10, fp, ip, lr}
+       mov fp, sp
+       sub sp, sp, #8
+@   for i := 0 to 9 do
+       mov r0, #0
+       set r1, _i
+       str r0, [r1]
+       mov r5, #9
+.L2:
+       set r0, _i
+       ldr r0, [r0]
+       cmp r0, r5
+       bgt .L3
+@     for j := 0 to 9 do
+       mov r0, #0
+       set r1, _j
+       str r0, [r1]
+       mov r4, #9
+.L4:
+       set r7, _j
+       ldr r8, [r7]
+       cmp r8, r4
+       bgt .L5
+@       a[i][j] := i + j
+       set r0, _i
+       ldr r9, [r0]
+       add r0, r9, r8
+       set r1, _a
+       mov r2, #40
+       mul r2, r9, r2
+       add r1, r1, r2
+       lsl r2, r8, #2
+       add r1, r1, r2
+       str r0, [r1]
+       ldr r0, [r7]
+       add r0, r0, #1
+       str r0, [r7]
+       b .L4
+.L5:
+@   end;
+       set r7, _i
+       ldr r0, [r7]
+       add r0, r0, #1
+       str r0, [r7]
+       b .L2
+.L3:
+@   for i := 0 to 9 do
+       mov r0, #0
+       set r1, _i
+       str r0, [r1]
+       mov r0, #9
+       str r0, [fp, #-4]
+.L6:
+       set r0, _i
+       ldr r0, [r0]
+       ldr r1, [fp, #-4]
+       cmp r0, r1
+       bgt .L1
+@     for j := 0 to 9 do
+       mov r0, #0
+       set r1, _j
+       str r0, [r1]
+       mov r6, #9
+.L8:
+       set r7, _j
+       ldr r8, [r7]
+       cmp r8, r6
+       bgt .L9
+@       print_num(a[i][j]);
+       set r0, _a
+       set r1, _i
+       ldr r1, [r1]
+       mov r2, #40
+       mul r1, r1, r2
+       add r0, r0, r1
+       lsl r1, r8, #2
+       add r0, r0, r1
+       ldr r0, [r0]
+       bl print_num
+@       print_char(' ')
+       mov r0, #32
+       bl print_char
+       ldr r0, [r7]
+       add r0, r0, #1
+       str r0, [r7]
+       b .L8
+.L9:
+@     newline()
+       bl newline
+       set r7, _i
+       ldr r0, [r7]
+       add r0, r0, #1
+       str r0, [r7]
+       b .L6
+.L1:
+       ldmfd fp, {r4-r10, fp, sp, pc}
+       .ltorg
+
+        .comm _a, 400, 4
+        .comm _i, 4, 4
+        .comm _j, 4, 4
+@ End
+]]*)
--- /dev/null
+++ b/test/range_pointer.p
@@ -0,0 +1,95 @@
+var a: array 10 of pointer to integer;
+var i, sum: integer;
+
+begin
+  i := 0;
+  for r in a do
+    new(r);
+    r^ := i;
+    i := i+1
+  end;
+  sum := 0;
+  for r in a do
+    sum := sum + r^
+  end;
+  print_num(sum);
+  newline()
+end.
+
+(*<<
+45
+>>*)
+
+(*[[
+@ picoPascal compiler output
+        .include "fixup.s"
+        .global pmain
+
+        .text
+pmain:
+        mov ip, sp
+        stmfd sp!, {r4-r10, fp, ip, lr}
+        mov fp, sp
+        sub sp, sp, #8
+@   i := 0;
+        mov r0, #0
+        set r1, _i
+        str r0, [r1]
+@   for r in a do
+        set r4, _a
+        add r5, r4, #40
+        b .L3
+.L2:
+@     new(r);
+        mov r0, #4
+        bl new
+        str r0, [r4]
+@     r^ := i;
+        set r7, _i
+        ldr r0, [r7]
+        ldr r1, [r4]
+        str r0, [r1]
+@     i := i+1
+        ldr r0, [r7]
+        add r0, r0, #1
+        str r0, [r7]
+        add r4, r4, #4
+.L3:
```

```
+       cmp r4, r5
+       bne .L2
+@   sum := 0;
+       mov r0, #0
+       set r1, _sum
+       str r0, [r1]
+@   for r in a do
+       set r6, _a
+       add r0, r6, #40
+       str r0, [fp, #-4]
+       b .L5
+.L4:
+@     sum := sum + r^
+       set r7, _sum
+       ldr r0, [r7]
+       ldr r1, [r6]
+       ldr r1, [r1]
+       add r0, r0, r1
+       str r0, [r7]
+       add r6, r6, #4
+.L5:
+       ldr r0, [fp, #-4]
+       cmp r6, r0
+       bne .L4
+@   print_num(sum);
+       set r0, _sum
+       ldr r0, [r0]
+       bl print_num
+@   newline()
+       bl newline
+       ldmfd fp, {r4-r10, fp, sp, pc}
+       .ltorg
+
+       .comm _a, 40, 4
+       .comm _i, 4, 4
+       .comm _sum, 4, 4
+@ End
+]]*)
--- /dev/null
+++ b/test/range_pointer2.p
@@ -0,0 +1,95 @@
+var p: pointer to array 10 of integer;
+var i, sum: integer;
+
+begin
+  new(p);
+  i := 0;
+  for r in p^ do
+    r := i;
+    i := i+1
+  end;
+  sum := 0;
+  for r in p^ do
+    sum := sum + r
+  end;
+  print_num(sum);
+  newline()
+end.
+
+(*<<
+45
+>>*)
+
+(*[[
+@ picoPascal compiler output
```

```
+       .include "fixup.s"
+       .global pmain
+
+       .text
+pmain:
+       mov ip, sp
+       stmfd sp!, {r4-r10, fp, ip, lr}
+       mov fp, sp
+       sub sp, sp, #8
+@   new(p);
+       mov r0, #40
+       bl new
+       set r1, _p
+       str r0, [r1]
+@   i := 0;
+       mov r1, #0
+       set r2, _i
+       str r1, [r2]
+@   for r in p^ do
+       mov r4, r0
+       add r5, r4, #40
+       b .L3
+.L2:
+@     r := i;
+       set r7, _i
+       ldr r0, [r7]
+       str r0, [r4]
+@     i := i+1
+       ldr r0, [r7]
+       add r0, r0, #1
+       str r0, [r7]
+       add r4, r4, #4
+.L3:
+       cmp r4, r5
+       bne .L2
+@   sum := 0;
+       mov r0, #0
+       set r1, _sum
+       str r0, [r1]
+@   for r in p^ do
+       set r0, _p
+       ldr r6, [r0]
+       add r0, r6, #40
+       str r0, [fp, #-4]
+       b .L5
+.L4:
+@     sum := sum + r
+       set r7, _sum
+       ldr r0, [r7]
+       ldr r1, [r6]
+       add r0, r0, r1
+       str r0, [r7]
+       add r6, r6, #4
+.L5:
+       ldr r0, [fp, #-4]
+       cmp r6, r0
+       bne .L4
+@   print_num(sum);
+       set r0, _sum
+       ldr r0, [r0]
+       bl print_num
+@   newline()
+       bl newline
+       ldmfd fp, {r4-r10, fp, sp, pc}
+       .ltorg
```

```
+
+       .comm _p, 4, 4
+       .comm _i, 4, 4
+       .comm _sum, 4, 4
+@ End
+]]*)
--- /dev/null
+++ b/test/range_pointer2_for.p
@@ -0,0 +1,98 @@
+var p: pointer to array 10 of integer;
+var i, sum: integer;
+
+begin
+  new(p);
+  for i := 0 to 9 do
+    p^[i] := i
+  end;
+  sum := 0;
+  for i := 0 to 9 do
+    sum := sum + p^[i]
+  end;
+  print_num(sum);
+  newline()
+end.
+
+(*<<
+45
+>>*)
+
+(*[[
+@ picoPascal compiler output
+       .include "fixup.s"
+       .global pmain
+
+       .text
+pmain:
+       mov ip, sp
+       stmfd sp!, {r4-r10, fp, ip, lr}
+       mov fp, sp
+@   new(p);
+       mov r0, #40
+       bl new
+       set r1, _p
+       str r0, [r1]
+@   for i := 0 to 9 do
+       mov r0, #0
+       set r1, _i
+       str r0, [r1]
+       mov r4, #9
+.L2:
+       set r6, _i
+       ldr r7, [r6]
+       cmp r7, r4
+       bgt .L3
+@     p^[i] := i
+       set r0, _p
+       ldr r0, [r0]
+       lsl r1, r7, #2
+       add r0, r0, r1
+       str r7, [r0]
+       ldr r0, [r6]
+       add r0, r0, #1
+       str r0, [r6]
+       b .L2
+.L3:
```

```
+@   sum := 0;
+       mov r0, #0
+       set r1, _sum
+       str r0, [r1]
+@   for i := 0 to 9 do
+       mov r0, #0
+       set r1, _i
+       str r0, [r1]
+       mov r5, #9
+.L4:
+       set r6, _i
+       ldr r7, [r6]
+       cmp r7, r5
+       bgt .L5
+@     sum := sum + p^[i]
+       set r8, _sum
+       ldr r0, [r8]
+       set r1, _p
+       ldr r1, [r1]
+       lsl r2, r7, #2
+       add r1, r1, r2
+       ldr r1, [r1]
+       add r0, r0, r1
+       str r0, [r8]
+       add r0, r7, #1
+       str r0, [r6]
+       b .L4
+.L5:
+@   print_num(sum);
+       set r0, _sum
+       ldr r0, [r0]
+       bl print_num
+@   newline()
+       bl newline
+       ldmfd fp, {r4-r10, fp, sp, pc}
+       .ltorg
+
+       .comm _p, 4, 4
+       .comm _i, 4, 4
+       .comm _sum, 4, 4
+@ End
+]]*)
--- /dev/null
+++ b/test/range_pointer_for.p
@@ -0,0 +1,101 @@
+var a: array 10 of pointer to integer;
+var i, sum: integer;
+
+begin
+  for i := 0 to 9 do
+    new(a[i]);
+    a[i]^ := i
+  end;
+  sum := 0;
+  for i := 0 to 9 do
+    sum := sum + a[i]^
+  end;
+  print_num(sum);
+  newline()
+end.
+
+(*<<
+45
+>>*)
+
+(*[[
+@ picoPascal compiler output
+       .include "fixup.s"
+       .global pmain
+
+       .text
+pmain:
+       mov ip, sp
+       stmfd sp!, {r4-r10, fp, ip, lr}
+       mov fp, sp
+@   for i := 0 to 9 do
+       mov r0, #0
+       set r1, _i
+       str r0, [r1]
+       mov r4, #9
+.L2:
+       set r6, _i
+       ldr r0, [r6]
+       cmp r0, r4
+       bgt .L3
+@     new(a[i]);
+       mov r0, #4
+       bl new
+       set r7, _a
+       ldr r1, [r6]
+       lsl r1, r1, #2
+       add r1, r7, r1
+       str r0, [r1]
+@     a[i]^ := i
+       ldr r8, [r6]
+       lsl r0, r8, #2
+       add r0, r7, r0
+       ldr r0, [r0]
+       str r8, [r0]
+       ldr r0, [r6]
+       add r0, r0, #1
+       str r0, [r6]
+       b .L2
+.L3:
+@   sum := 0;
+       mov r0, #0
+       set r1, _sum
+       str r0, [r1]
+@   for i := 0 to 9 do
+       mov r0, #0
+       set r1, _i
+       str r0, [r1]
+       mov r5, #9
+.L4:
+       set r6, _i
+       ldr r7, [r6]
+       cmp r7, r5
+       bgt .L5
+@     sum := sum + a[i]^
+       set r8, _sum
+       ldr r0, [r8]
+       set r1, _a
+       lsl r2, r7, #2
+       add r1, r1, r2
+       ldr r1, [r1]
+       ldr r1, [r1]
+       add r0, r0, r1
+       str r0, [r8]
+       add r0, r7, #1
+       str r0, [r6]
+       b .L4
+.L5:
+@   print_num(sum);
+       set r0, _sum
+       ldr r0, [r0]
+       bl print_num
+@   newline()
+       bl newline
+       ldmfd fp, {r4-r10, fp, sp, pc}
+       .ltorg
+
+       .comm _a, 40, 4
+       .comm _i, 4, 4
+       .comm _sum, 4, 4
+@ End
+]]*)
--- /dev/null
+++ b/test/range_record.p
@@ -0,0 +1,99 @@
+type point = record x, y: integer end;
+var a: array 10 of point;
+var i, sum: integer;
+
+begin
+  i := 0;
+  for r in a do
+    r.x := i mod 3;
+    r.y := i mod 5;
+    i := i+1
+  end;
+  sum := 0;
+  for r in a do
+    sum := sum + r.x * r.y
+  end;
+  print_num(sum);
+  newline()
+end.
+
+(*<<
+17
+>>*)
+
+(*[[
+@ picoPascal compiler output
+       .include "fixup.s"
+       .global pmain
+
+       .text
+pmain:
+       mov ip, sp
+       stmfd sp!, {r4-r10, fp, ip, lr}
+       mov fp, sp
+       sub sp, sp, #8
+@   i := 0;
+       mov r0, #0
+       set r1, _i
+       str r0, [r1]
+@   for r in a do
+       set r4, _a
+       add r5, r4, #80
+       b .L3
+.L2:
+@     r.x := i mod 3;
+       set r7, _i
+       mov r1, #3
+       ldr r0, [r7]
+       bl int_mod
+       str r0, [r4]
+@     r.y := i mod 5;
+       mov r1, #5
+       ldr r0, [r7]
+       bl int_mod
+       str r0, [r4, #4]
+@     i := i+1
+       ldr r0, [r7]
+       add r0, r0, #1
+       str r0, [r7]
+       add r4, r4, #8
+.L3:
+       cmp r4, r5
+       bne .L2
+@   sum := 0;
+       mov r0, #0
+       set r1, _sum
+       str r0, [r1]
+@   for r in a do
+       set r6, _a
+       add r0, r6, #80
+       str r0, [fp, #-4]
+       b .L5
+.L4:
+@     sum := sum + r.x * r.y
+       set r7, _sum
+       ldr r0, [r7]
+       ldr r1, [r6]
+       ldr r2, [r6, #4]
+       mul r1, r1, r2
+       add r0, r0, r1
+       str r0, [r7]
+       add r6, r6, #8
+.L5:
+       ldr r0, [fp, #-4]
+       cmp r6, r0
+       bne .L4
+@   print_num(sum);
+       set r0, _sum
+       ldr r0, [r0]
+       bl print_num
+@   newline()
+       bl newline
+       ldmfd fp, {r4-r10, fp, sp, pc}
+       .ltorg
+
+       .comm _a, 80, 4
+       .comm _i, 4, 4
+       .comm _sum, 4, 4
+@ End
+]]*)
--- /dev/null
+++ b/test/range_record_for.p
@@ -0,0 +1,106 @@
+type point = record x, y: integer end;
+var a: array 10 of point;
+var i, sum: integer;
+
+begin
+  for i := 0 to 9 do
+    a[i].x := i mod 3;
+    a[i].y := i mod 5
+  end;
+  sum := 0;
+  for i := 0 to 9 do
+    sum := sum + a[i].x * a[i].y
+  end;
+  print_num(sum);
+  newline()
+end.
+
+(*<<
+17
+>>*)
+
+(*[[
+@ picoPascal compiler output
+       .include "fixup.s"
+       .global pmain
+
+       .text
+pmain:
+       mov ip, sp
+       stmfd sp!, {r4-r10, fp, ip, lr}
+       mov fp, sp
+@   for i := 0 to 9 do
+       mov r0, #0
+       set r1, _i
+       str r0, [r1]
+       mov r4, #9
+.L2:
+       set r6, _i
+       ldr r7, [r6]
+       cmp r7, r4
+       bgt .L3
+@     a[i].x := i mod 3;
+       mov r1, #3
+       mov r0, r7
+       bl int_mod
+       set r7, _a
+       ldr r1, [r6]
+       lsl r1, r1, #3
+       add r1, r7, r1
+       str r0, [r1]
+@     a[i].y := i mod 5
+       mov r1, #5
+       ldr r0, [r6]
+       bl int_mod
+       ldr r1, [r6]
+       lsl r1, r1, #3
+       add r1, r7, r1
+       str r0, [r1, #4]
+       ldr r0, [r6]
+       add r0, r0, #1
+       str r0, [r6]
+       b .L2
+.L3:
+@   sum := 0;
+       mov r0, #0
+       set r1, _sum
+       str r0, [r1]
+@   for i := 0 to 9 do
+       mov r0, #0
+       set r1, _i
+       str r0, [r1]
+       mov r5, #9
+.L4:
+       set r6, _i
+       ldr r7, [r6]
+       cmp r7, r5
+       bgt .L5
+@      sum := sum + a[i].x * a[i].y
+       set r8, _sum
+       set r0, _a
+       lsl r1, r7, #3
+       add r9, r0, r1
+       ldr r0, [r8]
+       ldr r1, [r9]
+       ldr r2, [r9, #4]
+       mul r1, r1, r2
+       add r0, r0, r1
+       str r0, [r8]
+       add r0, r7, #1
+       str r0, [r6]
+       b .L4
+.L5:
+@   print_num(sum);
+       set r0, _sum
+       ldr r0, [r0]
+       bl print_num
+@   newline()
+       bl newline
+       ldmfd fp, {r4-r10, fp, sp, pc}
+       .ltorg
+
+       .comm _a, 80, 4
+       .comm _i, 4, 4
+       .comm _sum, 4, 4
+@ End
+]]*)
--- /dev/null
+++ b/test/range_ref.p
@@ -0,0 +1,107 @@
+var a: array 10 of integer;
+
+proc init(var a: array 10 of integer);
+  var i: integer;
+begin
+  i := 0;
+  for r in a do
+    r := i;
+    i := i+1
+  end
+end;
+
+proc find_sum(var a: array 10 of integer): integer;
+  var sum: integer;
+begin
+  sum := 0;
+  for r in a do
+    sum := sum + r
+  end;
+  return sum
+end;
+
+begin
+  init(a);
+  print_num(find_sum(a));
+  newline()
+end.
+
+(*<<
+45
```

```
+>>*)
+
+(*[[
+@ picoPascal compiler output
+       .include "fixup.s"
+       .global pmain
+
+@ proc init(var a: array 10 of integer);
+       .text
+_init:
+       mov ip, sp
+       stmfd sp!, {r0-r1}
+       stmfd sp!, {r4-r10, fp, ip, lr}
+       mov fp, sp
+@   i := 0;
+       mov r4, #0
+@   for r in a do
+       ldr r5, [fp, #40]
+       add r6, r5, #40
+       b .L3
+.L2:
+@     r := i;
+       str r4, [r5]
+@     i := i+1
+       add r4, r4, #1
+       add r5, r5, #4
+.L3:
+       cmp r5, r6
+       bne .L2
+       ldmfd fp, {r4-r10, fp, sp, pc}
+       .ltorg
+
+@ proc find_sum(var a: array 10 of integer): integer;
+_find_sum:
+       mov ip, sp
+       stmfd sp!, {r0-r1}
+       stmfd sp!, {r4-r10, fp, ip, lr}
+       mov fp, sp
+@   sum := 0;
+       mov r4, #0
+@   for r in a do
+       ldr r5, [fp, #40]
+       add r6, r5, #40
+       b .L6
+.L5:
+@     sum := sum + r
+       ldr r0, [r5]
+       add r4, r4, r0
+       add r5, r5, #4
+.L6:
+       cmp r5, r6
+       bne .L5
+@   return sum
+       mov r0, r4
+       ldmfd fp, {r4-r10, fp, sp, pc}
+       .ltorg
+
+pmain:
+       mov ip, sp
+       stmfd sp!, {r4-r10, fp, ip, lr}
+       mov fp, sp
+@   init(a);
+       set r4, _a
+       mov r0, r4
+       bl _init
```

```
+@   print_num(find_sum(a));
+       mov r0, r4
+       bl _find_sum
+       bl print_num
+@   newline()
+       bl newline
+       ldmfd fp, {r4-r10, fp, sp, pc}
+       .ltorg
+
+       .comm _a, 40, 4
+@ End
+]]*)
--- /dev/null
+++ b/test/range_ref_for.p
@@ -0,0 +1,108 @@
+var a: array 10 of integer;
+
+proc init(var a: array 10 of integer);
+  var i: integer;
+begin
+  for i := 0 to 9 do
+    a[i] := i
+  end
+end;
+
+proc find_sum(var a: array 10 of integer): integer;
+  var i, sum: integer;
+begin
+  sum := 0;
+  for i := 0 to 9 do
+    sum := sum + a[i]
+  end;
+  return sum
+end;
+
+begin
+  init(a);
+  print_num(find_sum(a));
+  newline()
+end.
+
+(*<<
+45
+>>*)
+
+(*[[
+@ picoPascal compiler output
+       .include "fixup.s"
+       .global pmain
+
+@ proc init(var a: array 10 of integer);
+       .text
+_init:
+       mov ip, sp
+       stmfd sp!, {r0-r1}
+       stmfd sp!, {r4-r10, fp, ip, lr}
+       mov fp, sp
+@   for i := 0 to 9 do
+       mov r4, #0
+       mov r5, #9
+.L2:
+       cmp r4, r5
+       bgt .L1
+@     a[i] := i
+       ldr r0, [fp, #40]
+       lsl r1, r4, #2
+       add r0, r0, r1
+       str r4, [r0]
+       add r4, r4, #1
+       b .L2
+.L1:
+       ldmfd fp, {r4-r10, fp, sp, pc}
+       .ltorg
+
+@ proc find_sum(var a: array 10 of integer): integer;
+_find_sum:
+       mov ip, sp
+       stmfd sp!, {r0-r1}
+       stmfd sp!, {r4-r10, fp, ip, lr}
+       mov fp, sp
+@   sum := 0;
+       mov r5, #0
+@   for i := 0 to 9 do
+       mov r4, #0
+       mov r6, #9
+.L5:
+       cmp r4, r6
+       bgt .L6
+@     sum := sum + a[i]
+       ldr r0, [fp, #40]
+       lsl r1, r4, #2
+       add r0, r0, r1
+       ldr r0, [r0]
+       add r5, r5, r0
+       add r4, r4, #1
+       b .L5
+.L6:
+@   return sum
+       mov r0, r5
+       ldmfd fp, {r4-r10, fp, sp, pc}
+       .ltorg
+
+pmain:
+       mov ip, sp
+       stmfd sp!, {r4-r10, fp, ip, lr}
+       mov fp, sp
+@   init(a);
+       set r4, _a
+       mov r0, r4
+       bl _init
+@   print_num(find_sum(a));
+       mov r0, r4
+       bl _find_sum
+       bl print_num
+@   newline()
+       bl newline
+       ldmfd fp, {r4-r10, fp, sp, pc}
+       .ltorg
+
+       .comm _a, 40, 4
+@ End
+]]*)
+
--- a/test/strcopy.p
+++ b/test/strcopy.p
@@ -1,13 +1,13 @@
 (* String copying by loop *)

-const in = "Hello, world!*";
+const ins = "Hello, world!*";
```

```
 var out: array 128 of char; i: integer;

 begin
   i := 0;
-  while in[i] <> '*' do
-    out[i] := in[i];
+  while ins[i] <> '*' do
+    out[i] := ins[i];
     i := i + 1
   end;
   out[i] := chr(0);
```

```
--- a/Makefile
+++ b/Makefile
@@ -5,7 +5,7 @@ all: ppc
 TOOLS = ../tools

 ppc: util.cmo mach.cmo optree.cmo dict.cmo tree.cmo lexer.cmo \
-               parser.cmo check.cmo target.cmo regs.cmo simp.cmo \
+               parser.cmo check.cmo com_check.cmo target.cmo regs.cmo simp.cmo \
                share.cmo jumpopt.cmo tran.cmo tgen.cmo main.cmo
        ocamlc -g ../lib/common.cma $^ -o $@

@@ -26,18 +26,8 @@ MLFLAGS = -I ../lib
 $(TOOLS)/nodexp $(TOOLS)/pibake: $(TOOLS)/%:
        $(MAKE) -C $(TOOLS) $*

-test: force
-       @echo "Say..."
-       @echo "  'make test0' to compare assembly code"
-       @echo "  'make test1' to test using QEMU"
-       @echo "  'make test2' to test using a remote RPi"
-       @echo "  'make test3' to test using ECSLAB remotely"
-
 TESTSRC := $(shell ls test/*.p)
-OPT = -O2
-
-SCRIPT1 = -e '1,/^(\*\[\[/d' -e '/^]]\*)/q' -e p
-SCRIPT2 = -e '1,/^(\*<</d' -e '/^>>\*)/q' -e p
+STRICT = -strictio

 ARMGCC = arm-linux-gnueabihf-gcc -marm -march=armv6

@@ -49,62 +39,38 @@ ifndef QEMU
     QEMU := qemu-arm
 endif

-# test0 -- compile tests and diff object code
-test0 : $(TESTSRC:test/%.p=test0-%)
-
-test0-%: force
-       @echo "*** Test $*.p"
-       ./ppc $(OPT) test/$*.p >b.s
-       -sed -n $(SCRIPT1) test/$*.p | diff -u -b - b.s
-       @echo
-
-# test1 -- compile tests and execute with QEMU
-test1 : $(TESTSRC:test/%.p=test1-%)
+# test -- compile tests
+test: $(TESTSRC:test/wrong_%.p=wrong-%) $(TESTSRC:test/right_%.p=right-%)

-test1-%: pas0.o force
-       @echo "*** Test $*.p"
-       ./ppc $(OPT) test/$*.p >b.s
-       $(ARMGCC) b.s pas0.o -static -o b.out
-       $(QEMU) ./b.out >b.test
-       sed -n $(SCRIPT2) test/$*.p | diff - b.test
-       @echo "*** Passed"; echo
+wrong-%: force
+       @echo "*** Test wrong_$*.p"
+       echo "Possibly incorrect" > b.ans
+       ./ppc test/wrong_$*.p | diff b.ans -

-pas0.o: pas0.c
-       $(ARMGCC) -c $< -o $@
+right-%: force
+       @echo "*** Test right_$*.p"
+       echo "Correct" > b.ans
+       -./ppc test/right_$*.p | diff b.ans -

-# test2 -- compile tests and execute using remote or local RPi
-test2 : $(TESTSRC:test/%.p=test2-%)
+# teststrict -- compile tests with -strictio
+teststrict: $(TESTSRC:test/wrong_%.p=wrongstrict-%) $(TESTSRC:test/right_%.p=rightstrict-%)

-test2-%: $(TOOLS)/pibake force
-       @echo "*** Test $*.p"
-       ./ppc $(OPT) test/$*.p >b.s
-       $(TOOLS)/pibake b.s >b.test
-       sed -n $(SCRIPT2) test/$*.p | diff - b.test
-       @echo "*** Passed"; echo
+wrongstrict-%: force
+       @echo "*** Test wrong_$*.p"
+       echo "Possibly incorrect" > b.ans
+       ./ppc $(STRICT) test/wrong_$*.p | diff b.ans -

-# test3 -- ditto but using qemu on ecs.ox
-test3 : $(TESTSRC:test/%.p=test3-%)
-
-test3-%: $(TOOLS)/ecsx force
-       @echo "*** Test $*.p"
-       ./ppc $(OPT) test/$*.p >b.s
-       $(TOOLS)/ecsx pas0.c fixup.s b.s >b.test
-       sed -n $(SCRIPT2) test/$*.p | diff - b.test
-       @echo "*** Passed"; echo
-
-promote: $(TESTSRC:test/%.p=promote-%)
-
-promote-%: force
-       ./ppc $(OPT) test/$*.p >b.s
-       sed -f promote.sed test/$*.p >test/$*.new
-       mv test/$*.new test/$*.p
+rightstrict-%: force
+       @echo "*** Test right_$*.p"
+       echo "Correct" > b.ans
+       -./ppc $(STRICT) test/right_$*.p | diff b.ans -

 force:

 MLGEN = parser.mli parser.ml lexer.ml

 ML = $(MLGEN) optree.ml tgen.ml tran.ml simp.ml share.ml jumpopt.ml \
-       check.ml check.mli dict.ml dict.mli lexer.mli \
+       check.ml check.mli com_check.ml com_check.mli dict.ml dict.mli lexer.mli \
        mach.ml mach.mli main.ml optree.mli tgen.mli tree.ml \
        tree.mli util.ml tran.mli target.mli target.ml \
        simp.mli share.mli regs.mli regs.ml jumpopt.mli
@@ -146,6 +112,9 @@ check.cmo : util.cmo tree.cmi optree.cmi mach.cmi lexer.cmi dict.cmi \
 check.cmx : util.cmx tree.cmx optree.cmx mach.cmx lexer.cmx dict.cmx \
     check.cmi
 check.cmi : tree.cmi
+com_check.cmo : util.cmo tree.cmi dict.cmi com_check.cmi
+com_check.cmx : util.cmx tree.cmx dict.cmx com_check.cmi
+com_check.cmi : tree.cmi
 dict.cmo : util.cmo optree.cmi mach.cmi dict.cmi
 dict.cmx : util.cmx optree.cmx mach.cmx dict.cmi
 dict.cmi : optree.cmi mach.cmi
@@ -154,9 +123,9 @@ mach.cmo : mach.cmi
 mach.cmx : mach.cmi
 mach.cmi :
 main.cmo : tree.cmi tran.cmi tgen.cmi parser.cmi mach.cmi lexer.cmi \
-    check.cmi
```

```
+    check.cmi com_check.cmi
 main.cmx : tree.cmx tran.cmx tgen.cmx parser.cmx mach.cmx lexer.cmx \
-    check.cmx
+    check.cmx com_check.cmx
 optree.cmi :
 tgen.cmi : tree.cmi
 tree.cmo : optree.cmi dict.cmi tree.cmi
--- a/check.ml
+++ b/check.ml
@@ -280,7 +280,7 @@ let check_const e env =

 let make_def x k t =
   { d_tag = x; d_kind = k; d_type = t; d_level = !level;
-    d_mem = false; d_addr = Nowhere }
+    d_mem = false; d_addr = Nowhere; d_node = -1 }

 (* check_dupcases -- check for duplicate case labels *)
 let check_dupcases vs =
@@ -301,6 +301,10 @@ let rec check_stmt s env alloc =
     | Seq ss ->
         List.iter (fun s1 -> check_stmt s1 env alloc) ss

+    | Commutative (s1, s2) ->
+        check_stmt s1 env alloc;
+        check_stmt s2 env alloc
+
     | Assign (lhs, rhs) ->
         let lt = check_expr lhs env
         and rt = check_expr rhs env in
@@ -571,8 +575,8 @@ and check_bodies env ds =
 (* INITIAL ENVIRONMENT *)

 let defn (x, k, t) env =
-  let d = { d_tag = intern x; d_kind = k; d_type = t;
-            d_level = 0; d_mem = false; d_addr = Nowhere } in
+  let d = { d_tag = intern x; d_kind = k; d_type = t; d_level = 0;
+            d_mem = false; d_addr = Nowhere; d_node = -1 } in
   define d env

 let libproc i n ts =
--- /dev/null
+++ b/com_check.ml
@@ -0,0 +1,770 @@
+open Tree
+open Dict
+open Util
+open Print
+
+(* Whether io effects should be strict *)
+let strict_io = ref false
+
+(* Used for disabling errors in preliminary passes *)
+let enable_errors = ref true
+
+(* |Com_error| -- exception raised on commutative error *)
+exception Com_error of string
+
+(* |com_error| -- report a commutative error and abandon analysis *)
+let com_error str = if !enable_errors then raise (Com_error str)
+
+(* |state| -- state of a variable after operations *)
+type state = Same | Overwritten | Partial | Invalid
+
+(* |effect| -- read, state -- whether it reads the previous value, in what state it is left *)
+type effect = bool * state
```

```
+
+(* |id_type| -- type for identifying objects with effects on them *)
+type id_type =
+    Var of int * int * ident        (* concrete object: type, level, tag *)
+  | Ref of int * int * ident        (* reference to object: type, level, tag *)
+  | Heap of int                     (* object in the heap: type *)
+  | Element of int * id_type        (* element of array: type, array *)
+  | Field of int * ident * id_type  (* field of record: type, field, record *)
+  | Return                          (* return value (non-void) *)
+  | IO                              (* anything affecting input and output *)
+
+module IdMap = Map.Make(struct
+    type t = id_type
+    let compare = compare
+  end)
+
+(* |effect_map| -- type for keeping track of affected objects *)
+type effect_map = effect IdMap.t
+
+(* |big_effect| -- noncommutative, returns, effects, unpredictable levels, conflicts *)
+type big_effect = bool * bool * effect_map * (int list) * ((id_type * id_type) list)
+
+(* |node| -- represents a node in the function dependancy graph *)
+type node =
+  {  n_index: int;                              (* Index of node *)
+     n_body: stmt;                              (* Function body *)
+     n_level: int;                              (* Function level *)
+     n_params: def list;                        (* Function parameters *)
+     mutable n_calls: int list;                 (* Functions this one calls *)
+     mutable n_called: int list;                (* Functions that call this one*)
+     mutable n_effect: big_effect option;       (* Effect of executing the function *)
+     mutable n_argeffs: effect_map option list; (* Effect of function on its arguments *)
+     mutable n_visit: bool;                     (* Helper boolean for DFS *)
+     mutable n_scc: int }                       (* Stores the index of the node's SCC *)
+
+(* Array of all nodes in the dependency graph *)
+let nodes = ref [||]
+
+(* Current node, used for building the graph *)
+let curr_node = ref 0
+
+(* Current level *)
+let level = ref 0
+
+(* Some helper function for states *)
+let same = function Same -> true | _ -> false
+let changed = function Same -> false | _ -> true
+let propagated = function Same | Partial -> true | _ -> false
+let overwritten = function Overwritten -> true | _ -> false
+let invalid = function Invalid -> true | _ -> false
+let valid = function Invalid -> false | _ -> true
+
+(* |seq_effs| -- composes two sequential effects *)
+let seq_effs (r1, s1) (r2, s2) =
+  if r2 && invalid s1 then com_error "Reading invalid\n";
+  let r = r1 || (r2 && not (overwritten s1)) in
+  let s = if same s2 then s1 else
+          if not (propagated s2) then s2 else
+          if changed s1 then s1 else
+          Partial in
+  (r, s)
+
+(* |com_effs| -- composes two commutative effects *)
+let com_effs (r1, s1) (r2, s2) =
+  if (r1 && changed s2) || (r2 && changed s1) then com_error "Commuting read and write\n";
+  let r = r1 || r2 in
+  let s = if invalid s1 || invalid s2 || (changed s1 && changed s2) then Invalid else
+          if overwritten s1 || overwritten s2 then Overwritten else
+          if changed s1 || changed s2 then Partial else
+          Same in
+  (r, s)
+
+(* |alt_effs| -- composes two alternative effects *)
+let alt_effs (r1, s1) (r2, s2) =
+  let r = r1 || r2 in
+  let s = if invalid s1 || invalid s2 then Invalid else
+          if overwritten s1 && overwritten s2 then Overwritten else
+          if same s1 && same s2 then Same else
+          Partial in
+  (r, s)
+
+(* |to_list| -- utility function for producing a list from a map *)
+let to_list em =
+  IdMap.fold (fun id eff xs -> (id, eff) :: xs) em []
+
+(* |to_list| -- utility function for producing a map from a list *)
+let from_list el =
+  List.fold_left (fun em (id, eff) -> IdMap.add id eff em) IdMap.empty el
+
+(* |combine_ems| -- combines two effect maps according to a rule *)
+let combine_ems rule em1 em2 =
+  let combine1 id eff1 em =
+    let eff2 = match IdMap.find_opt id em2 with
+        None -> (false, Same)
+      | Some eff -> eff in
+    let eff = rule eff1 eff2 in
+    IdMap.add id eff em
+  and combine2 id eff2 em =
+    match IdMap.find_opt id em1 with
+        None ->
+          let eff = rule (false, Same) eff2 in
+          IdMap.add id eff em
+      | Some eff -> em in
+  let temp = IdMap.fold combine1 em1 IdMap.empty in
+  IdMap.fold combine2 em2 temp
+
+(* |ref_match| -- checks whether two different ids could refer to the same object *)
+let rec ref_match id1 id2 =
+  let rec lev_match id lv =
+    match id with
+        Var (_, lv2, _) -> lv2 < lv
+      | Ref (_, _, _) -> true
+      | Heap _ -> true
+      | Element (_, id2) -> lev_match id2 lv
+      | Field (_, _, id2) -> lev_match id2 lv
+      | Return | IO -> failwith "lev_match" in
+  let type_lev_match id t lv =
+    match id with
+        Var (t2, lv2, _) -> t2 = t && lv2 < lv
+      | Ref (t2, _, _) -> failwith "type_lev_match"
+      | Heap t2 -> t2 = t
+      | Element (t2, id2) -> t2 = t && lev_match id lv
+      | Field (t2, _, id2) -> t2 = t && lev_match id lv
+      | Return | IO -> false in
+  match id1, id2 with
+      Ref (t1, lv1, x1), Ref (t2, lv2, x2) -> t1 = t2 && (x1 <> x2 || lv1 <> lv2)
+    | Ref (t, lv, _), _ -> type_lev_match id2 t lv
+    | _, Ref (t, lv, _) -> type_lev_match id1 t lv
+    | Element (t1, id12), Element (t2, id22) -> t1 = t2 && ref_match id12 id22
+    | Field (t1, x1, id12), Field (t2, x2, id22) -> t1 = t2 && x1 = x2 && ref_match id12 id22
+     | _ -> false
+
+(* |above| -- checks whether some id is above a level *)
+let rec above lvl id = match id with
+     Var (_, lvl2, _) -> lvl2 < lvl
+   | Ref _ -> lvl > 0
+   | Heap _ -> lvl > 0
+   | Element (_, id2) -> above lvl id2
+   | Field (_, _, id2) -> above lvl id2
+   | Return -> true
+   | IO -> true
+
+(* |transform_em_above| -- modifies all effects above some level *)
+let transform_em_above rule lvl em =
+   let accum id eff em =
+     let new_eff = if above lvl id then rule eff else eff in
+     IdMap.add id new_eff em in
+   IdMap.fold accum em IdMap.empty
+
+(* |partialize_em| -- turns all overwrites above lvl to partial overwrites *)
+let partialize_em_above lvl em =
+   let partialize (r, s) =
+     let s2 = if overwritten s then Partial else s in
+     (r, s2) in
+   transform_em_above partialize lvl em
+
+(* |invalidate_em| -- turns all changes above lvl to invalid states *)
+let invalidate_em_above lvl em =
+   let invalidate (r, s) =
+     let s2 = if changed s then Invalid else s in
+     (r, s2) in
+   transform_em_above invalidate lvl em
+
+(* |valid_above| -- confirms that all states above a level are valid *)
+let valid_above lvl em cf =
+   let check id (_, s) =
+     if invalid s && above lvl id then com_error "Invalid state before unpredictable call\n" in
+   if lvl > 0 then begin
+     if cf <> [] then com_error "Reference conflicts before unpredictable call\n";
+     IdMap.iter check em
+   end
+
+(* |max_level| -- utility function *)
+let max_level up =
+   match up with
+       [] -> -1
+     | x::xs -> List.fold_left max x xs
+
+(* |combine_big_seq| -- combines two sequential big_effects, demotes the second one if needed
    *)
+let combine_big_seq (nc1, rt1, em1, up1, cf1) (nc2, rt2, em2, up2, cf2) =
+   let keep_conflict (id1, id2) =
+     let to_keep effo = match effo with
+         None -> false
+       | Some (r, s) ->
+           if r then com_error "Reading conflict\n";
+           not (overwritten s || invalid s) in
+     let keep1 = to_keep (IdMap.find_opt id1 em2)
+     and keep2 = to_keep (IdMap.find_opt id2 em2) in
+     keep1 || keep2 in
+   let max_up = max_level up2 in
+   valid_above max_up em1 cf1;
+   let em22 = if rt1 then partialize_em_above !level em2 else em2
+   and cf12 = List.filter keep_conflict cf1 in
+   (nc1 || nc2, rt1 || rt2, combine_ems seq_effs em1 em22, List.concat [up1; up2], List.concat
   [cf12; cf2])
+
+(* |combine_big_com| -- combines two commutative big_effects, invalidates them if needed *)
+let combine_big_com (nc1, rt1, em1, up1, cf1) (nc2, rt2, em2, up2, cf2) =
+  let check_conflict em (id1, id2) =
+    let check_one effo = match effo with
+        Some (true, _) -> com_error "Reading conflict in commutative statement\n"
+      | _ -> () in
+    check_one (IdMap.find_opt id1 em);
+    check_one (IdMap.find_opt id2 em) in
+  let rec mini_find el id1 r1 s1 = match el with
+      [] -> []
+    | (id2, (r2, s2)) :: tail ->
+        let rest = mini_find tail id1 r1 s1 in
+        if (not r2 && same s2) || (not (ref_match id1 id2)) then rest
+        else begin
+          if (r1 && changed s2) || (r2 && changed s1) then com_error "Reference read-write
   commute confilict\n";
+          if changed s1 && valid s1 && changed s2 && valid s2 then (id1, id2) :: rest else
   rest
+        end in
+  let rec find_conflicts el = match el with
+      [] -> []
+    | (id, (r, s)) :: tail ->
+        let rest = find_conflicts tail in
+        if not r && same s then rest
+        else List.concat [mini_find (to_list em2) id r s; rest] in
+  let all_below lvl em =
+    let f id eff = if above lvl id then com_error "Clash in commuting unpredictable call\n" in
+    IdMap.iter f em in
+  if nc1 || nc2 then com_error "Commuting noncommutative\n";
+  let max_up1 = max_level up1
+  and max_up2 = max_level up2 in
+  let anyup1 = max_up1 > 0
+  and anyup2 = max_up2 > 0 in
+  if anyup1 || anyup2 then begin
+    if !strict_io then com_error "Commuting an unpredictable call\n";
+    if anyup1 && anyup2 then com_error "Commuting two unpredictable calls\n";
+    if (anyup1 && rt2) || (anyup2 && rt1) then com_error "Commuting unpredictable call with
   return\n";
+    if anyup1 then all_below max_up1 em2;
+    if anyup2 then all_below max_up1 em2
+  end;
+  List.iter (check_conflict em2) cf1;
+  List.iter (check_conflict em1) cf2;
+  let em12 = if rt2 then invalidate_em_above !level em1 else em1
+  and em22 = if rt1 then invalidate_em_above !level em2 else em2
+  and cf3 = find_conflicts (to_list em1) in
+  (nc1 || nc2, rt1 || rt2, combine_ems com_effs em12 em22, List.concat [up1; up2], List.concat
     [cf1; cf2; cf3])
+
+(* |combine_big_alt| -- combines two alternatives big_effects *)
+let combine_big_alt (nc1, rt1, em1, up1, cf1) (nc2, rt2, em2, up2, cf2) =
+  (nc1 || nc2, rt1 || rt2, combine_ems alt_effs em1 em2, List.concat [up1; up2], List.concat [
   cf1; cf2])
+
+(* |partialize_big| -- turns all overwrites to partial overwrites *)
+let partialize_big (nc, rt, em, up, cf) =
+  (nc, rt, partialize_em_above (!level + 1) em, up, cf)
+
+(* Functions for generating effects for a single id *)
+
+(* |expand| -- handles records and arrays *)
+let rec expand id t =
+  match t.t_guts with
+      BasicType _ -> [id]
+    | PointerType _ -> [id]
+    | ArrayType (_, t2) -> expand (Element (t2.t_id, id)) t2
+    | RecordType ds ->
+        let expand1 d = expand (Field (d.d_type.t_id, d.d_tag, id)) d.d_type in
+        List.concat (List.map expand1 ds)
+    | _ -> failwith "expand"
+
+(* |em_effect| -- generates an effect from an effect map *)
+let em_effect em = (false, false, em, [], [])
+
+(* |no_effect| -- operation without an observable effect *)
+let no_effect = em_effect IdMap.empty
+
+(* |noncommutative_effect| -- operation which never commutes *)
+let noncommutative_effect = (true, false, IdMap.empty, [], [])
+
+(* |return_void_effect| -- operation which returns void *)
+let return_void_effect = (false, true, IdMap.empty, [], [])
+
+(* |return_value_effect| -- operation which returns a value *)
+let return_value_effect = (false, true, IdMap.singleton Return (false, Overwritten), [], [])
+
+(* |io_effect| -- operation which prints, reads, etc. *)
+let io_effect = em_effect (IdMap.singleton IO (false, Partial))
+
+(* |unpredictable_effect| -- function call with unpredictable effects *)
+let unpredictable_effect lvl = (false, false, IdMap.empty, [lvl], [])
+
+(* |read_effect| -- operation which reads the values of the ids *)
+let read_effect id t =
+  let ids = expand id t in
+  let effs = List.map (fun id -> (id, (true, Same))) ids in
+  em_effect (from_list effs)
+
+(* |partial_effect| -- operation which maybe writes to the ids *)
+let partial_effect id t =
+  let ids = expand id t in
+  let effs = List.map (fun id -> (id, (false, Partial))) ids in
+  em_effect (from_list effs)
+
+(* |write_effect| -- operation which writes to the ids *)
+let write_effect id t =
+  let rec state id = match id with
+      Var _ -> Overwritten
+    | Ref _ -> Overwritten
+    | Heap _ -> Partial
+    | Element _ -> Partial
+    | Field (_, _, id2) -> state id2
+    | _ -> failwith "write_effect" in
+  let ids = expand id t in
+  let effs = List.map (fun id -> (id, (false, state id))) ids in
+  em_effect (from_list effs)
+
+(* |get| -- gets the value of an option *)
+let get o =
+  match o with
+      Some x -> x
+    | None -> failwith "get"
+
+(* |exists| -- check whether an option has a value *)
+let exists o =
+  match o with
+      Some _ -> true
+    | None -> false
```

```
+
+(* |rep| -- replicates x n times *)
+let rec rep x n =
+  if n <= 0 then [] else x :: rep x (n - 1)
+
+(* |strictify| -- adds an noncommutative_effect if io is strict *)
+let strictify eff =
+  if !strict_io then combine_big_alt eff noncommutative_effect
+  else eff
+
+(* |reroot| -- reroots all ids and splits the em into a read effect and a write effect *)
+let reroot id t emo =
+  let rec reroot_one arg_id = match arg_id with
+      Ref _ -> id
+    | Element (t, id2) -> Element (t, reroot_one id2)
+    | Field (t, x, id2) -> Field (t, x, reroot_one id2)
+    | Var _ | Heap _ | Return | IO -> failwith "reroot_one" in
+  let accum arg_id (r, s) (rem, wem) =
+    let new_id = reroot_one arg_id in
+    let new_rem = if r then IdMap.add new_id (r, Same) rem else rem
+    and new_wem = if changed s then IdMap.add new_id (false, s) wem else wem in
+    (new_rem, new_wem) in
+  if exists emo then
+    let (rem, wem) = IdMap.fold accum (get emo) (IdMap.empty, IdMap.empty) in
+    (em_effect rem, em_effect wem)
+  else (read_effect id t, partial_effect id t)
+
+(* |check_arg| -- check one (formal, actual) parameter pair *)
+let rec check_arg formal argeff arg =
+  match formal.d_kind with
+      CParamDef -> (check_expr arg, no_effect)
+    | VParamDef ->
+        let (eff, ido) = check_id arg in
+        let (read_eff, write_eff) = reroot (get ido) formal.d_type argeff in
+        (combine_big_seq eff read_eff, write_eff)
+    | PParamDef ->
+        let d = match arg.e_guts with
+            Variable x -> get x.x_def
+          | _ -> failwith "check_arg" in
+        begin match d.d_kind with
+            ProcDef -> (partialize_big (get !nodes.(d.d_node).n_effect), no_effect)
+          | PParamDef -> (unpredictable_effect d.d_level, no_effect)
+          | _ -> failwith "check_arg"
+        end
+    | _ -> failwith "check_arg"
+
+(* |check_args| -- match formal and actual parameters *)
+and check_args formals argeffs args =
+  let rec map3 f xs ys zs = match xs, ys, zs with
+      x::xs, y::ys, z::zs -> f x y z :: map3 f xs ys zs
+    | _ -> [] in
+  let cmb (r_eff1, w_eff1) (r_eff2, w_eff2) =
+    let r_eff3 = combine_big_seq r_eff1 r_eff2
+    and w_eff3 = combine_big_seq w_eff1 w_eff2 in
+    (r_eff3, w_eff3) in
+  let rw_effs = map3 check_arg formals argeffs args in
+  List.fold_left cmb (no_effect, no_effect) rw_effs
+
+(* |check_libcall| -- check a library function or procedure call *)
+and check_libcall q args =
+  let strict_io = strictify io_effect in
+  match (q.q_id, args) with
+      (ChrFun, [e]) -> check_expr e
+    | (OrdFun, [e]) -> check_expr e
+    | (PrintNum, [e]) -> combine_big_alt (check_expr e) strict_io
+    | (PrintChar, [e]) -> combine_big_alt (check_expr e) strict_io
+    | (PrintString, [e]) -> combine_big_alt (check_expr e) strict_io
+    | (NewLine, []) -> strict_io
+    | (ReadChar, [e]) -> combine_big_alt (check_write e) strict_io
+    | (ExitProc, [e]) -> strictify (check_expr e)
+    | (NewProc, [e]) -> strictify (check_write e)
+    | (ArgcFun, []) -> no_effect
+    | (ArgvProc, [e1; e2]) -> strictify (combine_big_seq (check_expr e1) (check_write e2))
+    | (OpenIn, [e]) -> combine_big_alt (check_expr e) strict_io
+    | (CloseIn, []) -> strict_io
+    | (Operator op, [e1]) -> check_expr e1
+    | (Operator op, [e1; e2]) ->
+        let eff = combine_big_seq (check_expr e1) (check_expr e2) in
+        begin match op with
+            Div | Mod ->
+              begin match e2.e_value with
+                  None | Some 0 -> strictify eff
+                | Some _ -> eff
+              end
+          | _ -> eff
+        end
+    | _ -> failwith "check_libcall"
+
+(* |check_funcall| -- check a function or procedure call *)
+and check_funcall f args =
+  let d = get f.x_def in
+  match d.d_kind with
+      LibDef q -> check_libcall q args
+    | ProcDef | PParamDef ->
+        let p = get_proc d.d_type in
+        let params = p.p_fparams in
+        let func_eff, arg_effs = match d.d_kind with
+            ProcDef -> (get !nodes.(d.d_node).n_effect, !nodes.(d.d_node).n_argeffs)
+          | PParamDef -> (unpredictable_effect d.d_level, rep None (List.length params))
+          | _ -> failwith "check_funcall" in
+        let (rarg_eff, warg_eff) = check_args params arg_effs args in
+        let eff1 = combine_big_seq rarg_eff func_eff in
+        combine_big_seq eff1 warg_eff
+    | _ -> failwith "check_funcall"
+
+(* |check_write| -- check a write to an identity *)
+and check_write e =
+  let (eff, ido) = check_id e in
+  combine_big_seq eff (write_effect (get ido) e.e_type)
+
+(* |check_id| -- check an expression for an identity *)
+and check_id e =
+  match e.e_guts with
+      Variable x ->
+        let d = get x.x_def in
+        let ido = match d.d_kind with
+            VarDef | CParamDef -> Some (Var (e.e_type.t_id, d.d_level, d.d_tag))
+          | VParamDef -> Some (Ref (e.e_type.t_id, d.d_level, d.d_tag))
+          | ConstDef _ | StringDef -> None
+          | _ -> failwith "check_id" in
+        (no_effect, ido)
+
+    | Sub (e1, e2) ->
+        let (eff1, ido) = check_id e1
+        and eff2 = check_expr e2 in
+        let ido2 = match ido with
+            None -> None
+          | Some id -> Some (Element (e.e_type.t_id, id)) in
+        let tot_eff = combine_big_seq eff1 eff2 in
+        begin match e2.e_value with
+             None -> (strictify tot_eff, ido2)
+           | Some n ->
+               if n >= 0 && n < bound e1.e_type then (tot_eff, ido2)
+               else (strictify tot_eff, ido2)
+         end
+
+     | Select (e1, x) ->
+         let (eff1, ido) = check_id e1 in
+         let ido2 = match ido with
+             None -> None
+           | Some id -> Some (Field (e.e_type.t_id, x.x_name, id)) in
+         (eff1, ido2)
+
+     | Deref e1 ->
+         let eff1 = check_expr e1 in
+         let ido = Some (Heap e.e_type.t_id) in
+         (strictify (eff1), ido)
+
+     | _ -> (check_expr e, None)
+
+(* |check_expr| -- check an expression *)
+and check_expr e =
+  match e.e_guts with
+     Variable _ | Sub _ | Select _ | Deref _ ->
+         let (eff, ido) = check_id e in
+         if not (exists ido) then eff
+         else combine_big_seq eff (read_effect (get ido) e.e_type)
+     | Constant _ | String _ | Nil -> no_effect
+     | FuncCall (p, args) -> check_funcall p args
+     | Monop (w, e1) -> check_expr e1
+     | Binop (w, e1, e2) ->
+         let eff1 = check_expr e1
+         and eff2 = check_expr e2 in
+         combine_big_seq eff1 eff2
+
+(* |check_stmt| -- check a statement *)
+and check_stmt s =
+  match s.s_guts with
+       Skip -> no_effect
+
+     | Seq ss ->
+         let effs = List.map (fun s1 -> check_stmt s1) ss in
+         List.fold_left combine_big_seq no_effect effs
+
+     | Commutative (s1, s2) ->
+         let eff1 = check_stmt s1
+         and eff2 = check_stmt s2 in
+         combine_big_com eff1 eff2
+
+     | Assign (lhs, rhs) ->
+         let (eff1, ido) = check_id lhs
+         and eff2 = check_expr rhs in
+         let expr_eff = combine_big_seq eff1 eff2 in
+         combine_big_seq expr_eff (write_effect (get ido) lhs.e_type)
+
+     | ProcCall (p, args) -> check_funcall p args
+
+     | Return res ->
+         if not (exists res) then return_void_effect
+         else combine_big_seq (check_expr (get res)) return_value_effect
+
+     | IfStmt (cond, thenpt, elsept) ->
+         let cond_eff = check_expr cond
+         and then_eff = check_stmt thenpt
+         and else_eff = check_stmt elsept in
+         let body_eff = combine_big_alt then_eff else_eff in
+         combine_big_seq cond_eff body_eff
+
+     | WhileStmt (cond, body) ->
+         let cond_eff = check_expr cond
+         and body_eff = check_stmt body in
+         let maybe_eff = partialize_big body_eff in
+         combine_big_seq cond_eff maybe_eff
+
+     | RepeatStmt (body, test) ->
+         let body_eff = check_stmt body
+         and test_eff = check_expr test in
+         combine_big_seq body_eff test_eff
+
+     | ForStmt (var, lo, hi, body, _) ->
+         let var_eff = check_write var
+         and lo_eff = check_expr lo
+         and hi_eff = check_expr hi
+         and body_eff = check_stmt body in
+         let init_eff = combine_big_seq lo_eff var_eff in
+         let for_eff = combine_big_seq init_eff hi_eff
+         and maybe_eff = partialize_big body_eff in
+         combine_big_seq for_eff maybe_eff
+
+     | CaseStmt (sel, arms, deflt) ->
+         let check_arm (lab, body) =
+           check_stmt body in
+         let sel_eff = check_expr sel
+         and arm_effs = List.map check_arm arms
+         and deflt_eff = check_stmt deflt in
+         let case_effs = List.fold_left combine_big_alt deflt_eff arm_effs in
+         combine_big_seq sel_eff case_effs
+
+(* |keep_above| -- keeps the effects above a scope and checks they are valid *)
+let keep_above lvl (nc, rt, em, up, cf) =
+  let rec keep_one id =
+    match id with
+        Var (_, lvl2, _) -> (lvl2 < lvl, lvl2 < lvl)
+      | Ref (_, lvl2, _) -> (true, lvl2 < lvl)
+      | Heap _ -> (lvl > 0, lvl > 0)
+      | Element (_, id2) -> keep_one id2
+      | Field (_, _, id2) -> keep_one id2
+      | Return -> (true, false)
+      | IO -> (true, true) in
+  let one_iter id (_, s) =
+    let (keep_valid, keep) = keep_one id in
+    if invalid s && keep_valid then com_error "Invalid state at return\n";
+    keep
+  and keep_unpred lv = lv < lvl in
+  if cf <> [] then com_error "Reference conflicts at return\n";
+  (nc, false, IdMap.filter one_iter em, List.filter keep_unpred up, [])
+
+(* |get_argeffs| -- extracts the effects to each argument only *)
+let get_argeffs params (_, _, em, _, _) =
+  let rec is_match lvl x id = match id with
+      Var _ | Heap _ | Return | IO -> false
+    | Ref (_, lvl2, x2) -> lvl2 = lvl && x2 = x
+    | Element (_, id2) -> is_match lvl x id2
+    | Field (_, _, id2) -> is_match lvl x id2 in
+  let get_match lvl x id eff em =
+    if is_match lvl x id then IdMap.add id eff em
+    else em in
+  let get_one p = match p.d_kind with
+      VParamDef -> Some (IdMap.fold (get_match p.d_level p.d_tag) em IdMap.empty)
+    | CParamDef | PParamDef -> None
+    | _ -> failwith "get_argeffs" in
+  List.map get_one params
+
+(* |check_node| -- checks a single function body *)
+let check_node update_argeffs n =
+  level := n.n_level;
+  let eff = check_stmt n.n_body in
+  n.n_effect <- Some (keep_above n.n_level eff);
+  if update_argeffs then
+    n.n_argeffs <- get_argeffs n.n_params eff
+
+(* |check_scc| -- checks a the nodes in a single non-empty scc *)
+let check_scc scc =
+  let ns = List.map (fun x -> !nodes.(x)) scc in
+  let root = List.hd ns in
+  if List.length ns = 1 && not (List.mem root.n_index root.n_calls) then
+     check_node true root
+  else begin
+    let assign_effect eff n =
+      n.n_effect <- Some (keep_above n.n_level eff)
+    and clean_effect (nc, rt, em, up, cf) =
+      let clean (id, (_, s)) = if overwritten s then [id, (false, s)] else [] in
+      (false, rt, from_list (List.concat (List.map clean (to_list em))), [], []) in
+    enable_errors := false;
+    List.iter (assign_effect no_effect) ns;
+    let clean_one n =
+      n.n_effect <- Some (clean_effect (get n.n_effect)) in
+    List.iter (check_node false) ns;
+    List.iter clean_one ns;
+    List.iter (check_node false) ns;
+    let effs = List.map (fun n -> get n.n_effect) ns in
+    let tot_eff = List.fold_left combine_big_alt (List.hd effs) (List.tl effs) in
+    List.iter (assign_effect tot_eff) ns;
+    enable_errors := true;
+    List.iter (check_node true) ns
+  end
+
+(* |kosaraju_check| -- find scc-s in rev. top. order and solve *)
+let kosaraju_check num_nodes =
+  let stack = ref [] in
+  let rec visit x =
+    let n = !nodes.(x) in
+    if not n.n_visit then begin
+      n.n_visit <- true;
+      List.iter visit n.n_called;
+      stack := x :: !stack
+    end in
+  for x = 0 to num_nodes - 1 do
+    visit x
+  done;
+  let rec assign root x =
+    let n = !nodes.(x) in
+    if n.n_scc = - 1 then begin
+      n.n_scc <- root;
+      x :: List.concat (List.map (assign root) n.n_calls);
+    end else [] in
+  let handle x =
+    if !nodes.(x).n_scc = -1 then check_scc (assign x x) in
+  List.iter handle !stack
+
+(* |make_node| -- creates a node *)
+let make_node i s l ps =
+  { n_index = i; n_body = s; n_level = l; n_params = ps;  n_calls = []; n_called = [];
+    n_effect = None; n_argeffs = rep None (List.length ps); n_visit = false; n_scc = -1; }
+
+(* |add_dependency| -- adds a call edge from x to y *)
+let add_dependency x y =
+  if not (List.mem y !nodes.(x).n_calls) then
+    !nodes.(x).n_calls <- y :: !nodes.(x).n_calls;
+    !nodes.(y).n_called <- x :: !nodes.(y).n_called
+
+(* |graph_arg| -- build graph at one argument *)
+let rec graph_arg formal arg =
+  match formal.d_kind with
+      CParamDef | VParamDef -> graph_expr arg
+    | PParamDef ->
+        let other_node = match arg.e_guts with
+            Variable x -> (get x.x_def).d_node
+          | _ -> failwith "graph_arg" in
+        if other_node >= 0 then add_dependency !curr_node other_node
+    | _ -> failwith "graph_arg"
+
+(* |graph_args| -- match formal and actual parameters *)
+and graph_args formals args =
+  List.iter2 (fun f a -> graph_arg f a) formals args
+
+(* |graph_funcall| -- builds graph at function call *)
+and graph_funcall f args =
+  let d = get f.x_def in
+  match d.d_kind with
+      LibDef _ -> List.iter graph_expr args;
+    | PParamDef ->
+        let p = get_proc d.d_type in
+        graph_args p.p_fparams args
+    | ProcDef ->
+        let p = get_proc d.d_type in
+        graph_args p.p_fparams args;
+        let other_node = d.d_node in
+        add_dependency !curr_node other_node
+    | _ -> failwith "graph_funcall"
+
+(* |graph_expr| -- builds graph at expression *)
+and graph_expr e =
+  match e.e_guts with
+      Variable x -> ()
+    | Sub (e1, e2) -> graph_expr e1; graph_expr e2
+    | Select (e1, x) -> graph_expr e1
+    | Deref e1 -> graph_expr e1
+    | Constant _ | String _ | Nil -> ()
+    | FuncCall (p, args) -> graph_funcall p args
+    | Monop (_, e1) -> graph_expr e1
+    | Binop (_, e1, e2) -> graph_expr e1; graph_expr e2
+
+(* |graph_stmt| -- builds graph at statement *)
+and graph_stmt s =
+  match s.s_guts with
+      Skip -> ()
+    | Seq ss -> List.iter graph_stmt ss
+    | Commutative (s1, s2) -> graph_stmt s1; graph_stmt s2
+    | Assign (lhs, rhs) -> graph_expr lhs; graph_expr rhs
+    | ProcCall (p, args) -> graph_funcall p args
+    | Return res -> if exists res then graph_expr (get res)
+    | IfStmt (cond, thenpt, elsept) ->
+        graph_expr cond; graph_stmt thenpt; graph_stmt elsept
+    | WhileStmt (cond, body) -> graph_expr cond; graph_stmt body
+    | RepeatStmt (body, test) -> graph_stmt body; graph_expr test
+    | ForStmt (var, lo, hi, body, _) ->
+        graph_expr var; graph_expr lo;
+        graph_expr hi; graph_stmt body
+    | CaseStmt (sel, arms, deflt) ->
+        graph_expr sel;
+        List.iter (fun (lab, body) -> graph_stmt body) arms;
+        graph_stmt deflt
+
+(* |graph_node| -- builds graph at node *)
+and graph_node n =
+  curr_node := n.n_index;
+  graph_stmt n.n_body
+
+(* |build_nodes| -- builds the nodes array *)
+let rec build_nodes ds =
+  let build_one d =
+    match d with
+        ProcDecl (Heading(x, _, _), (Block (ds, ss, _, _))) ->
+          let fd = get_def x in
+          let params = (get_proc fd.d_type).p_fparams in
+          !nodes.(!curr_node) <- make_node !curr_node ss (fd.d_level + 1) params;
+          fd.d_node <- !curr_node;
+          incr curr_node;
+          build_nodes ds
+      |  _ -> () in
+  List.iter build_one ds
+
+(* |count_funcs| -- counts the number of functions declared *)
+let rec count_funcs ds =
+  let accum_one sum d =
+    match d with
+        ProcDecl (_, (Block (ds, _, _, _))) ->
+          sum + 1 + count_funcs ds
+      |  _ -> sum in
+  List.fold_left accum_one 0 ds
+
+(* |com_check| -- check tree for commutative errors *)
+let com_check (Prog (Block (globals, ss, _, _), _)) =
+  let num_nodes = 1 + count_funcs globals in
+  nodes := Array.make num_nodes (make_node 0 ss 0 []);
+  incr curr_node;
+  build_nodes globals;
+  Array.iter graph_node !nodes;
+  kosaraju_check num_nodes
--- /dev/null
+++ b/com_check.mli
@@ -0,0 +1,10 @@
+open Tree
+
+(* |com_check| -- check tree for commutative errors *)
+val com_check : program -> unit
+
+(* Exception raised when a commutative error is detected *)
+exception Com_error of string
+
+(* |strict_io| -- whether io effects should be strict *)
+val strict_io : bool ref
--- a/dict.ml
+++ b/dict.ml
@@ -94,7 +94,8 @@ and def =
     d_type: ptype;              (* Type *)
     d_level: int;               (* Static level *)
     mutable d_mem: bool;        (* Whether addressible *)
-    mutable d_addr: location }  (* Run-time location *)
+    mutable d_addr: location;   (* Run-time location *)
+    mutable d_node: int }       (* Node in the function dependency graph (if it is a function)
     *)

 and basic_type = VoidType | IntType | CharType | BoolType | AddrType
```

```
--- a/dict.mli
+++ b/dict.mli
@@ -57,7 +57,8 @@ and def =
     d_type: ptype;              (* Type *)
     d_level: int;               (* Static level *)
     mutable d_mem: bool;        (* Whether addressible *)
-    mutable d_addr: location }  (* Run-time location *)
+    mutable d_addr: location;   (* Run-time location *)
+    mutable d_node: int }       (* Node in the function dependency graph (if it is a function)
     *)

 and basic_type = VoidType | IntType | CharType | BoolType | AddrType

--- a/main.ml
+++ b/main.ml
@@ -19,7 +19,9 @@ let spec =
       "-O2", Arg.Unit (fun () -> Tgen.optlevel := 2),
         " more optimisation (common subexpressions)";
       "-noregvars", Arg.Unit (fun () -> Check.regvars := false),
-        " disable register variables"]
+        " disable register variables";
+      "-strictio", Arg.Unit (fun () -> Com_check.strict_io := true),
+        " take a stricter approach to io commutativity"]

 let main () =
   let fns = ref [] in
@@ -56,8 +58,13 @@ let main () =
       exit 1
   end;

-  (* Translate the program *)
-  Tgen.translate prog;
+  (* Effect analysis *)
+  begin try Com_check.com_check prog with
+    Com_check.Com_error str ->
+      printf "Possibly incorrect\n" [];
+      exit 0
+  end;
+  printf "Correct\n" [];
   exit 0

 let ppc = main ()
--- a/parser.mly
+++ b/parser.mly
@@ -110,6 +110,7 @@ line :

 stmt1 :
     /* empty */                         { Skip }
+  | SUB stmts COLON stmts BUS           { Commutative ($2, $4) }
  | variable ASSIGN expr                { Assign ($1, $3) }
  | name actuals                        { ProcCall ($1, $2) }
  | RETURN expr_opt                     { Return $2 }
--- a/tgen.ml
+++ b/tgen.ml
@@ -257,6 +257,8 @@ let rec gen_stmt s =

      | Seq ss -> <SEQ, @(List.map gen_stmt ss)>

+     | Commutative (s1, s2) -> <SEQ, gen_stmt s1, gen_stmt s2>
+
      | Assign (v, e) ->
          if scalar v.e_type || is_pointer v.e_type then begin
            let st = if size_of v.e_type = 1 then STOREC else STOREW in
--- a/tree.ml
```

```
+++ b/tree.ml
@@ -31,6 +31,7 @@ and stmt =
 and stmt_guts =
     Skip
   | Seq of stmt list
+  | Commutative of stmt * stmt
   | Assign of expr * expr
   | ProcCall of name * expr list
   | Return of expr option
@@ -135,6 +136,7 @@ and fStmt s =
   match s.s_guts with
       Skip -> fStr "(SKIP)"
     | Seq stmts -> fMeta "(SEQ$)" [fTail(fStmt) stmts]
+    | Commutative (s1, s2) -> fMeta "COMMUTATIVE $ $" [fStmt s1; fStmt s2]
     | Assign (e1, e2) -> fMeta "(ASSIGN $ $)" [fExpr e1; fExpr e2]
     | ProcCall (p, aps) -> fMeta "(CALL $$)" [fName p; fTail(fExpr) aps]
     | Return (Some e) -> fMeta "(RETURN $)" [fExpr e]
--- a/tree.mli
+++ b/tree.mli
@@ -46,6 +46,7 @@ and stmt =
 and stmt_guts =
     Skip
   | Seq of stmt list
+  | Commutative of stmt * stmt
   | Assign of expr * expr
   | ProcCall of name * expr list
   | Return of expr option
@@ -95,3 +96,5 @@ val makeBlock : decl list * stmt -> block

 (* |print_tree| -- pretty-print a tree *)
 val print_tree : out_channel -> string -> program -> unit
+
+val fStmt : stmt -> Print.arg
```

```
--- /dev/null
+++ b/test/right_array.p
@@ -0,0 +1,9 @@
+var a, b: array 10 of integer;
+var i, j: integer;
+begin
+  i := 3;
+  j := 3;
+  [ a[i] := 2 : b[j] := 7 ];
+  print_num(a[3]); newline();
+  print_num(b[3]); newline()
+end.
--- /dev/null
+++ b/test/right_array2.p
@@ -0,0 +1,10 @@
+type point = record x, y: integer end;
+var a: array 10 of point;
+var i, j: integer;
+begin
+  i := 3;
+  j := 3;
+  [ a[i].x := 2 : a[j].y := 7 ];
+  print_num(a[3].x); newline();
+  print_num(a[3].y); newline()
+end.
--- /dev/null
+++ b/test/right_basic_procs.p
@@ -0,0 +1,16 @@
+var x, y: integer;
+
+proc f();
+begin
+  print_num(x); newline()
+end;
+
+proc g();
+begin
+  [ y := x : f() ];
+  print_num(y); newline()
+end;
+
+begin
+  g()
+end.
--- /dev/null
+++ b/test/right_case.p
@@ -0,0 +1,12 @@
+var x: integer;
+var y: integer;
+begin
+  [ x := 4 : x := 8 ];
+  case y of
+      6: x := 5
+    | 12: x := 3
+  else x := 2
+  end;
+  print_num(x); newline();
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/right_for.p
@@ -0,0 +1,10 @@
+var x: integer;
+var y: integer;
+begin
```

```
+  [ x := 4 : x := 8 ];
+  for x := 2 to 4 do
+    y := x
+  end;
+  print_num(x); newline();
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/right_heapread.p
@@ -0,0 +1,8 @@
+var p: pointer to integer;
+var x, y: integer;
+begin
+  new(p);
+  [ x := p^ : y := p^ ];
+  print_num(x); newline();
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/right_heaprecords.p
@@ -0,0 +1,12 @@
+var p, q: pointer to record x, y, z: integer end;
+begin
+  new(p);
+  q := p;
+  [ p^.x := 5 : q^.y := p^.z ];
+  print_num(p^.x); newline();
+  print_num(p^.y); newline();
+  print_num(p^.z); newline();
+  print_num(q^.x); newline();
+  print_num(q^.y); newline();
+  print_num(q^.z); newline()
+end.
--- /dev/null
+++ b/test/right_heaptypes.p
@@ -0,0 +1,9 @@
+var p: pointer to integer;
+var q: pointer to char;
+begin
+  new(p);
+  new(q);
+  [ p^ := 5 : q^ := 'd' ];
+  print_num(p^); newline();
+  print_char(q^); newline()
+end.
--- /dev/null
+++ b/test/right_if.p
@@ -0,0 +1,9 @@
+var x: integer;
+var y: integer;
+begin
+  [ x := 1 : x := 2 ];
+  if y = 5 then x := 2
+  else [ x := 4 : y := 2 ] end;
+  print_num(x); newline();
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/right_init_point.p
@@ -0,0 +1,12 @@
+type Point = record x, y: integer end;
+var p: Point;
+proc init(var p: Point);
+begin
+  [p.x := 0 : p.y := 2 ]
```

```
+end;
+
+begin
+  init(p);
+  print_num(p.x); newline();
+  print_num(p.y); newline()
+end.
--- /dev/null
+++ b/test/right_invalid.p
@@ -0,0 +1,6 @@
+var x: integer;
+begin
+  [ x := 1 : x := 2 ];
+  x := 3;
+  print_num(x); newline()
+end.
--- /dev/null
+++ b/test/right_invalid2.p
@@ -0,0 +1,9 @@
+var x: integer;
+var y: integer;
+begin
+  [ x := 1 : x := 2 ];
+  if y = 5 then x := 2
+  else [ x := 4 : y:= 2] end;
+  print_num(x); newline();
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/right_new.p
@@ -0,0 +1,6 @@
+var p, q: pointer to integer;
+begin
+  [ new(p); p^ := 2 : new(q) ];
+  print_num(p^); newline();
+  print_num(q^); newline()
+end.
--- /dev/null
+++ b/test/right_next_two.p
@@ -0,0 +1,13 @@
+var x: integer;
+proc next_two(x: integer);
+  var y, z: integer;
+begin
+  [ y := x + 1 : z := x + 2 ];
+  print_num(y); newline();
+  print_num(z); newline()
+end;
+
+begin
+  next_two(x);
+  print_num(x); newline()
+end.
--- /dev/null
+++ b/test/right_oref.p
@@ -0,0 +1,22 @@
+type Point = record x, y: integer end;
+var point: Point;
+
+proc print_x(var p: Point);
+begin
+  print_num(p.x); newline()
+end;
+
+proc increment_y(var p: Point; x: integer);
```

```
+begin
+  p.y := p.y + x
+end;
+
+proc sum_xy(var p: Point): integer;
+begin
+  return p.x + p.y
+end;
+
+begin
+  [ print_x(point) : increment_y(point, 5) ];
+  print_num(sum_xy(point)); newline()
+end.
--- /dev/null
+++ b/test/right_pparam.p
@@ -0,0 +1,18 @@
+var x: integer;
+
+proc g(y: integer);
+begin
+  print_num(x); newline();
+  x := y
+end;
+
+proc f(proc h(y: integer));
+  var a: integer;
+begin
+  [ a := 2 : a := x ];
+  h(8)
+end;
+
+begin
+  f(g);
+end.
--- /dev/null
+++ b/test/right_pparam2.p
@@ -0,0 +1,18 @@
+var x: integer;
+
+proc g(y: integer);
+begin
+  print_num(x); newline();
+  x := y
+end;
+
+proc f(proc h(y: integer));
+  var a: integer;
+begin
+  [ a := 2 : h(8) ];
+  print_num(a); newline()
+end;
+
+begin
+  f(g);
+end.
--- /dev/null
+++ b/test/right_proc.p
@@ -0,0 +1,12 @@
+var x: integer;
+
+proc f();
+  var a: integer;
+begin
+  [ a := 5 : a := x ]
+end;
```

```
+
+begin
+  f();
+  print_num(x); newline()
+end.
--- /dev/null
+++ b/test/right_proc2.p
@@ -0,0 +1,13 @@
+var x: integer;
+
+proc f();
+  var a: integer;
+begin
+  [ x := 5 : x := a ];
+  x := 3
+end;
+
+begin
+  f();
+  print_num(x); newline()
+end.
--- /dev/null
+++ b/test/right_proc3.p
@@ -0,0 +1,12 @@
+var x: integer;
+var y: integer;
+
+proc f();
+begin
+  y := x
+end;
+
+begin
+  [ f() : print_num(x); newline() ];
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/right_proc4.p
@@ -0,0 +1,12 @@
+var x: integer;
+var y: integer;
+
+proc f();
+begin
+  print_num(x); newline()
+end;
+
+begin
+  [ y := x : f() ];
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/right_read.p
@@ -0,0 +1,7 @@
+var x, y, z: integer;
+begin
+  [ y := x + 1 : z := x + 2 ];
+  print_num(x); newline();
+  print_num(y); newline();
+  print_num(z); newline()
+end.
--- /dev/null
+++ b/test/right_record.p
@@ -0,0 +1,6 @@
+var p: record x, y: integer end;
```

```
+begin
+  [ p.x := 2 : p.y := 7 ];
+  print_num(p.x); newline();
+  print_num(p.y); newline()
+end.
--- /dev/null
+++ b/test/right_record2.p
@@ -0,0 +1,7 @@
+var p, q: record x, y: integer end;
+begin
+  [ p := q : p.x := 7 ];
+  p.x := 12;
+  print_num(p.x); newline();
+  print_num(p.y); newline()
+end.
--- /dev/null
+++ b/test/right_record3.p
@@ -0,0 +1,8 @@
+var p, q: record x, y: integer end;
+begin
+  [ q.y := p.x : p.y := q.x ];
+  print_num(p.x); newline();
+  print_num(p.y); newline();
+  print_num(q.x); newline();
+  print_num(q.y); newline()
+end.
--- /dev/null
+++ b/test/right_recur.p
@@ -0,0 +1,15 @@
+proc fib(n: integer): integer;
+  var x, y: integer;
+begin
+  case n of
+      0: return 0
+    | 1: return 1
+  else
+    [ x := fib(n-1) : y := fib(n-2) ];
+    return x + y
+  end
+end;
+
+begin
+  print_num(fib(10)); newline()
+end.
--- /dev/null
+++ b/test/right_recur2.p
@@ -0,0 +1,32 @@
+proc foo(x, y: integer): integer;
+  var t, p: integer;
+  proc bar(z: integer): integer;
+    var q: integer;
+  begin
+    if z <= 0 then
+      return 0
+    else
+      [
+        q := foo(z * 2, z - 1)
+      :
+        y := y + z
+      ];
+      return q
+    end
+  end;
+begin
+  if (x <= 0) or (y <= 0) then
+    return 0
+  else
+    [
+      t := foo(x - 1, x)
+    :
+      p := bar(x + y)
+    ]
+  end;
+  return y + t + p
+end;
+
+begin
+  print_num(foo(3, 5)); newline()
+end.
--- /dev/null
+++ b/test/right_recur3.p
@@ -0,0 +1,34 @@
+var x, y: integer;
+
+proc foo(z: integer);
+begin
+  [ bar(z); y := x : x := 2 ];
+  print_num(y); newline();
+  x := 0
+end;
+
+proc bar(z: integer);
+begin
+  if z <= 0 then x := 5; return
+  else foo(z - 1); x := z end
+end;
+
+proc buzz(z: integer);
+begin
+  if z <= 0 then x := 5; return
+  else fizz(z - 1); x := z end
+end;
+
+proc fizz(z: integer);
+begin
+  [ buzz(z); y := x : x := 2 ];
+  print_num(y); newline();
+  x := 0
+end;
+
+begin
+  foo(5);
+  print_num(x); newline();
+  fizz(10);
+  print_num(x); newline();
+end.
--- /dev/null
+++ b/test/right_recur4.p
@@ -0,0 +1,34 @@
+var x: integer;
+
+proc foo(z: integer);
+begin
+  [ x := 1 : x := 2 ];
+  bar(z);
+  print_num(x); newline()
+end;
+
+proc bar(z: integer);
+begin
```

```
+  if z <= 0 then x := 5; return
+  else foo(z - 1); x := z end
+end;
+
+proc buzz(z: integer);
+begin
+  if z <= 0 then x := 5; return
+  else fizz(z - 1); x := z end
+end;
+
+proc fizz(z: integer);
+begin
+  [ x := 1 : x := 2 ];
+  buzz(z);
+  print_num(x); newline()
+end;
+
+begin
+  foo(5);
+  print_num(x); newline();
+  fizz(10);
+  print_num(x); newline();
+end.
--- /dev/null
+++ b/test/right_ref.p
@@ -0,0 +1,14 @@
+var x: integer;
+
+proc f(var y: integer);
+  var a: integer;
+begin
+  [ a := 5 : y := 3 ];
+  print_num(a); newline();
+  print_num(x); newline();
+  print_num(y); newline()
+end;
+
+begin
+  f(x)
+end.
--- /dev/null
+++ b/test/right_ref2.p
@@ -0,0 +1,13 @@
+var x: integer;
+
+proc f(var y: integer);
+begin
+  [ x := 5 : y := 3 ];
+  x := 1;
+  print_num(x); newline();
+  print_num(y); newline()
+end;
+
+begin
+  f(x)
+end.
--- /dev/null
+++ b/test/right_ref3.p
@@ -0,0 +1,13 @@
+var x: integer;
+
+proc f(var y: integer);
+begin
+  [ x := 5 : y := 3 ];
+  y := 1;
+  print_num(x); newline();
+  print_num(y); newline()
+end;
+
+begin
+  f(x)
+end.
--- /dev/null
+++ b/test/right_ref4.p
@@ -0,0 +1,14 @@
+var x: integer;
+var c: char;
+
+proc f(var y: integer);
+begin
+  [ y := 5 : c := 'f' ];
+  print_num(x); newline();
+  print_num(y); newline();
+  print_char(c); newline()
+end;
+
+begin
+  f(x)
+end.
--- /dev/null
+++ b/test/right_ref_nested.p
@@ -0,0 +1,18 @@
+var x: integer;
+proc f(var y: integer);
+  proc g();
+    var a: integer;
+    proc h();
+    begin
+      [ a := 2 : y := 8 ]
+    end;
+  begin
+    h(); print_num(a)
+  end;
+begin
+  g(); print_num(y)
+end;
+
+begin
+  f(x)
+end.
--- /dev/null
+++ b/test/right_ref_until.p
@@ -0,0 +1,14 @@
+var a: integer;
+proc test(var a: integer; n: integer);
+begin
+  [ a := 5 : a := 2 ];
+  repeat
+    a := n;
+    n := n * n
+  until n >= 100
+end;
+
+begin
+  test(a, 101);
+  print_num(a); newline()
+end.
--- /dev/null
+++ b/test/right_refrecord.p
@@ -0,0 +1,15 @@
```

```
+type point = record x, y: integer end;
+var p: point;
+
+proc f(var q: point);
+begin
+  [ p.x := 5 : q.y := 3 ];
+  print_num(p.x); newline();
+  print_num(p.y); newline();
+  print_num(q.x); newline();
+  print_num(q.y); newline()
+end;
+
+begin
+  f(p)
+end.
--- /dev/null
+++ b/test/right_refrecord2.p
@@ -0,0 +1,19 @@
+type point = record x, y: integer end;
+var p: point;
+var x: integer;
+
+proc f(var q: point);
+  var a: integer;
+begin
+  [ q.y := q.x; q.x := 5 : a := x; x := 2 ];
+  print_num(a); newline();
+  print_num(x); newline();
+  print_num(p.x); newline();
+  print_num(p.y); newline();
+  print_num(q.x); newline();
+  print_num(q.y); newline()
+end;
+
+begin
+  f(p)
+end.
--- /dev/null
+++ b/test/right_repeat.p
@@ -0,0 +1,11 @@
+var x: integer;
+var y: integer;
+begin
+  [ x := 1 : x := 2 ];
+  repeat
+    y := y + 1;
+    x := y
+  until y > 4;
+  print_num(x); newline();
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/right_repoint.p
@@ -0,0 +1,7 @@
+var p, q: pointer to integer;
+var x: integer;
+begin
+  new(p);
+  [ p^ := 4 : q := p ];
+  print_num(q^); newline()
+end.
--- /dev/null
+++ b/test/right_return.p
@@ -0,0 +1,8 @@
+proc f(x: integer): integer;
```

```
+begin
+  [ return 4 : x := 7 ]
+end;
+
+begin
+  print_num(f(6)); newline();
+end.
--- /dev/null
+++ b/test/right_return2.p
@@ -0,0 +1,15 @@
+proc f(x, y: integer);
+begin
+  [
+    if x <= 0 then return end
+  :
+    if y <= 0 then return end
+  ];
+  print_num(x); newline();
+  print_num(y); newline();
+  f(x - 1, y - 1)
+end;
+
+begin
+  f(12, 27)
+end.
--- /dev/null
+++ b/test/right_return3.p
@@ -0,0 +1,15 @@
+proc f(x, y: integer);
+begin
+  [
+    if x <= 0 then return end
+  :
+    y := y * 2
+  ];
+  print_num(x); newline();
+  print_num(y); newline();
+  f(x - 1, y - 1)
+end;
+
+begin
+  f(12, 27)
+end.
--- /dev/null
+++ b/test/right_return_write.p
@@ -0,0 +1,14 @@
+var a: integer;
+var succ: boolean;
+proc positive_cube(x: integer): boolean;
+  var res: integer;
+begin
+  [ if x <= 0 then return false end
+  : res := x * x * x ];
+  print_num(res); newline();
+  return true
+end;
+
+begin
+  succ := positive_cube(a)
+end.
--- /dev/null
+++ b/test/right_swaps.p
@@ -0,0 +1,23 @@
+var x, y, z, w: integer;
+
+proc twoswaps();
+  var a: integer;
+begin
+  [
+    a := x;
+    x := y;
+    y := a
+  :
+    a := z;
+    z := w;
+    w := a
+  ]
+end;
+
+begin
+  twoswaps();
+  print_num(x); newline();
+  print_num(y); newline();
+  print_num(z); newline();
+  print_num(w); newline()
+end.
--- /dev/null
+++ b/test/right_weird_oref.p
@@ -0,0 +1,18 @@
+type Point = record x, y: integer end;
+var point: Point;
+
+proc print_x(var p: Point);
+begin print_num(p.x); newline() end;
+
+proc increment_y(var p: Point; x: integer);
+begin p.y := p.y + x end;
+
+proc weird(var p: Point): integer;
+begin
+  [ print_x(point) : increment_y(point, 5) ];
+  return p.x + p.y
+end;
+
+begin
+  print_num(weird(point)); newline()
+end.
--- /dev/null
+++ b/test/right_while.p
@@ -0,0 +1,11 @@
+var x: integer;
+var y: integer;
+begin
+  [
+    while x > 3 do x := x - 1 end
+  :
+    while y < 2 do y := y + 1 end
+  ];
+  print_num(x); newline();
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/wrong_array.p
@@ -0,0 +1,8 @@
+var a: array 10 of integer;
+var i, j: integer;
+begin
+  i := 3;
+  j := 3;
+  [ a[i] := 2 : a[j] := 7 ];
+  print_num(a[3]); newline()
+end.
--- /dev/null
+++ b/test/wrong_array2.p
@@ -0,0 +1,9 @@
+var a: array 10 of integer;
+var i, j: integer;
+begin
+  i := 3;
+  j := 3;
+  [ a[i] := 2 : a[j] := 7 ];
+  a[0] := 0;
+  print_num(a[3]); newline()
+end.
--- /dev/null
+++ b/test/wrong_array_index.p
@@ -0,0 +1,10 @@
+var a: array 10 of integer;
+proc setij(i, j: integer);
+begin
+  [a[i] := 5 : a[j] := 2 ]
+end;
+
+begin
+  setij(2, 3);
+  print_num(a[0]); newline();
+end.
--- /dev/null
+++ b/test/wrong_case.p
@@ -0,0 +1,11 @@
+var x: integer;
+var y: integer;
+begin
+  [ x := 4 : x := 8 ];
+  case y of
+      6 : x := 5
+    | 12 : x := 3
+  end;
+  print_num(x); newline();
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/wrong_case2.p
@@ -0,0 +1,12 @@
+var x: integer;
+var y: integer;
+begin
+  [ x := 4 : x := 8 ];
+  case y of
+      6 : x := 5
+    | 12 : y := x; x := 3
+  else x := 2
+  end;
+  print_num(x); newline();
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/wrong_for.p
@@ -0,0 +1,11 @@
+var x: integer;
+var y: integer;
+var z: integer;
+begin
+  [ x := 1 : x := 2 ];
+  for y := z to 4 do
+    x := y
+  end;
+  print_num(x); newline();
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/wrong_for2.p
@@ -0,0 +1,14 @@
+var x: integer;
+var y: integer;
+var z: integer;
+begin
+  [
+    x := 8
+  :
+    for x := 2 to 4 do
+      y := x
+    end
+  ];
+  print_num(x); newline();
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/wrong_global.p
@@ -0,0 +1,11 @@
+var x: integer;
+
+proc print_x();
+begin
+  print_num(x); newline()
+end;
+
+begin
+  [ x := 3 : x := 2 ];
+  print_x()
+end.
--- /dev/null
+++ b/test/wrong_heap_while.p
@@ -0,0 +1,16 @@
+type ptr = pointer to integer;
+var p: ptr;
+proc test(p: ptr; n: integer);
+begin
+  [ p^ := 5 : p^ := 2 ];
+  while n < 100 do
+    p^ := n;
+    n := n * n
+  end
+end;
+
+begin
+  new(p);
+  test(p, 101);
+  print_num(p^); newline()
+end.
--- /dev/null
+++ b/test/wrong_heapinvalid.p
@@ -0,0 +1,8 @@
+var p, q: pointer to integer;
+var x: integer;
+begin
+  new(p);
+  q := p;
+  [ p^ := 4 : q^ := 7 ];
+  print_num(p^); newline()
```

```
+end.
--- /dev/null
+++ b/test/wrong_heapinvalid2.p
@@ -0,0 +1,9 @@
+var p, q: pointer to integer;
+var x: integer;
+begin
+  new(p);
+  new(q);
+  [ p^ := 4 : p^ := 7 ];
+  q^ := 5;
+  print_num(p^); newline()
+end.
--- /dev/null
+++ b/test/wrong_heapread.p
@@ -0,0 +1,8 @@
+var p, q: pointer to integer;
+var x: integer;
+begin
+  new(p);
+  q := p;
+  [ x := p^ : q^ := 7 ];
+  print_num(x); newline()
+end.
--- /dev/null
+++ b/test/wrong_heaprecords.p
@@ -0,0 +1,10 @@
+var p, q: pointer to record x, y: integer end;
+begin
+  new(p);
+  q := p;
+  [ p^.x := 5 : q^.y := p^.x ];
+  print_num(p^.x); newline();
+  print_num(p^.y); newline();
+  print_num(q^.x); newline();
+  print_num(q^.y); newline()
+end.
--- /dev/null
+++ b/test/wrong_if.p
@@ -0,0 +1,9 @@
+var x: integer;
+var y: integer;
+begin
+  [ x := 1 : x := 2 ];
+  if y = 5 then y := 5
+  else [ x := 4 : y:= 2] end;
+  print_num(x); newline();
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/wrong_invalid.p
@@ -0,0 +1,5 @@
+var x: integer;
+begin
+  [ x := 1 : x := 2 ];
+  print_num(x); newline()
+end.
--- /dev/null
+++ b/test/wrong_io.p
@@ -0,0 +1,3 @@
+begin
+  [ print_string("Hello ") : print_string("world!\n") ];
+end.
--- /dev/null
+++ b/test/wrong_nested.p
```

```
@@ -0,0 +1,11 @@
+var x: integer;
+var y: integer;
+begin
+  [
+    [ x := 1 : x := 2 ]
+  :
+    y := x
+  ];
+  print_num(x); newline();
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/wrong_nested2.p
@@ -0,0 +1,11 @@
+var x: integer;
+var y: integer;
+begin
+  [
+    [ x := 1 : x := 2 ]
+  :
+    x := y
+  ];
+  print_num(x); newline();
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/wrong_new.p
@@ -0,0 +1,6 @@
+var p, q: pointer to integer;
+begin
+  [ new(p); p^ := 2 : q := p ];
+  print_num(p^); newline();
+  print_num(q^); newline()
+end.
--- /dev/null
+++ b/test/wrong_oref.p
@@ -0,0 +1,22 @@
+type Point = record x, y: integer end;
+var point: Point;
+
+proc print_y(var p: Point);
+begin
+  print_num(p.y); newline()
+end;
+
+proc increment_y(var p: Point; x: integer);
+begin
+  p.y := p.y + x
+end;
+
+proc sum_xy(var p: Point): integer;
+begin
+  return p.x + p.y
+end;
+
+begin
+  [ print_y(point) : increment_y(point, 5) ];
+  print_num(sum_xy(point)); newline()
+end.
--- /dev/null
+++ b/test/wrong_oref_sumreset.p
@@ -0,0 +1,16 @@
+type Point = record x, y: integer end;
+var point: Point;
```

```
+
+proc print_sum(var p: Point);
+begin
+  print_num(p.y + p.x); newline()
+end;
+
+proc sum_reset(var p: Point);
+begin
+  [ print_sum(p) : p.x := 0]
+end;
+
+begin
+  sum_reset(point)
+end.
--- /dev/null
+++ b/test/wrong_pparam.p
@@ -0,0 +1,16 @@
+var x: integer;
+
+proc g();
+begin
+  x := 2
+end;
+
+proc f(proc h());
+begin
+  [ h() : x := 2 ]
+end;
+
+begin
+  f(g);
+  print_num(x); newline()
+end.
--- /dev/null
+++ b/test/wrong_pparam2.p
@@ -0,0 +1,18 @@
+var x: integer;
+
+proc g();
+begin
+  print_num(x); newline()
+end;
+
+proc f(proc h());
+begin
+  [ x := 4 : x := 2 ];
+  h();
+  x := 0
+end;
+
+begin
+  f(g);
+  print_num(x); newline()
+end.
--- /dev/null
+++ b/test/wrong_pparam3.p
@@ -0,0 +1,18 @@
+var x: integer;
+
+proc g(y: integer);
+begin
+  print_num(x); newline();
+  x := y
+end;
+
+proc f(proc h(y: integer));
+  var a: integer;
+begin
+  [ a := 2 : a := x ];
+  h(8)
+end;
+
+begin
+  [ f(g) : print_string("Hello World!") ]
+end.
--- /dev/null
+++ b/test/wrong_pparam_usefunc.p
@@ -0,0 +1,13 @@
+proc use_func(proc h(y: integer));
+begin
+  h(5)
+end;
+
+proc print_x(x: integer);
+begin
+  print_num(x); newline()
+end;
+
+begin
+  [ print_string("Hello!\n") : use_func(print_x) ]
+end.
--- /dev/null
+++ b/test/wrong_printproc.p
@@ -0,0 +1,8 @@
+proc print_x(x: integer);
+begin
+  print_num(x); newline()
+end;
+
+begin
+  [ print_x(5) : print_x(2) ];
+end.
--- /dev/null
+++ b/test/wrong_proc.p
@@ -0,0 +1,12 @@
+var x: integer;
+
+proc f();
+  var a: integer;
+begin
+  [ x := 5 : x := a ]
+end;
+
+begin
+  f();
+  print_num(x); newline()
+end.
--- /dev/null
+++ b/test/wrong_proc2.p
@@ -0,0 +1,14 @@
+var x: integer;
+
+proc f();
+  var a: integer;
+begin
+  [ x := 5 : x := a ];
+  return;
+  x := 3
+end;
+
+begin
+  f();
+  print_num(x); newline()
+end.
--- /dev/null
+++ b/test/wrong_proc3.p
@@ -0,0 +1,10 @@
+var x: integer;
+
+proc f();
+begin
+  x := 3
+end;
+
+begin
+  [ f() : print_num(x); newline() ]
+end.
--- /dev/null
+++ b/test/wrong_proc4.p
@@ -0,0 +1,10 @@
+var x: integer;
+
+proc f();
+begin
+  print_string("Hello world!\n")
+end;
+
+begin
+  [ f() : print_num(x); newline() ]
+end.
--- /dev/null
+++ b/test/wrong_proc5.p
@@ -0,0 +1,10 @@
+var x: integer;
+
+proc f();
+begin
+  print_num(x); newline()
+end;
+
+begin
+  [ f() : x := 2 ]
+end.
--- /dev/null
+++ b/test/wrong_read.p
@@ -0,0 +1,6 @@
+var x, y: integer;
+begin
+  [ x := 1 : y := x ];
+  print_num(x); newline();
+  print_num(y); newline()
+end.
--- /dev/null
+++ b/test/wrong_record.p
@@ -0,0 +1,6 @@
+var p, q: record x, y: integer end;
+begin
+  [ p := q : p.x := 7 ];
+  print_num(p.x); newline();
+  print_num(p.y); newline()
+end.
--- /dev/null
+++ b/test/wrong_record2.p
@@ -0,0 +1,8 @@
+var p, q: record x, y: integer end;
```

```
+begin
+  [ q.y := 2 : p := q ];
+  print_num(p.x); newline();
+  print_num(p.y); newline();
+  print_num(q.x); newline();
+  print_num(q.y); newline()
+end.
--- /dev/null
+++ b/test/wrong_recur.p
@@ -0,0 +1,23 @@
+proc foo(x, y: integer): integer;
+  var t, p: integer;
+  proc bar(z: integer);
+  begin
+    if z <= 0 then return end;
+    if foo(z, z + 1) > 0 then goo() end
+  end;
+  proc goo();
+  begin
+    x := 2
+  end;
+begin
+  if (x <= 0) or (y <= 0) then
+    return 0
+  else
+    [ t := foo(y - 2, x) : bar(x + y) ]
+  end;
+  return t
+end;
+
+begin
+  print_num(foo(3, 5)); newline()
+end.
--- /dev/null
+++ b/test/wrong_recur2.p
@@ -0,0 +1,14 @@
+var x: integer;
+proc foo(y: integer);
+  proc bar(z: integer);
+  begin
+    foo(z * 2)
+  end;
+begin
+  [ x := y : bar(y + 4) ]
+end;
+
+begin
+  foo(3);
+  print_num(x); newline()
+end.
--- /dev/null
+++ b/test/wrong_recur3.p
@@ -0,0 +1,13 @@
+proc foo(x: integer);
+  proc bar(y: integer);
+  begin
+    foo(x + y)
+  end;
+begin
+  [ foo(x + 1) : bar(x) ];
+  print_num(x); newline()
+end;
+
+begin
+  foo(3);
```

```
+end.
--- /dev/null
+++ b/test/wrong_ref.p
@@ -0,0 +1,11 @@
+var x: integer;
+
+proc f(var y: integer);
+begin
+  [ x := 5 : y := 3 ];
+  print_num(x); newline()
+end;
+
+begin
+  f(x)
+end.
--- /dev/null
+++ b/test/wrong_ref2.p
@@ -0,0 +1,11 @@
+var x: integer;
+
+proc f(var y: integer);
+begin
+  [ x := 5 : y := 3 ];
+  print_num(y); newline()
+end;
+
+begin
+  f(x)
+end.
--- /dev/null
+++ b/test/wrong_ref3.p
@@ -0,0 +1,12 @@
+var x: integer;
+
+proc f(var y: integer);
+  var a: integer;
+begin
+  [ a := x : y := 3 ];
+  print_num(a); newline()
+end;
+
+begin
+  f(x)
+end.
--- /dev/null
+++ b/test/wrong_ref4.p
@@ -0,0 +1,11 @@
+var x: integer;
+
+proc f(var y: integer);
+begin
+  [ x := 1 : y := 3 ]
+end;
+
+begin
+  f(x);
+  print_num(x); newline()
+end.
--- /dev/null
+++ b/test/wrong_ref5.p
@@ -0,0 +1,12 @@
+var x: integer;
+
+proc f(var y: integer);
+  var a, b: integer;
```

```
+begin
+  [ a := x : [ x := 1 : y := 3 ] ];
+  print_num(a); newline()
+end;
+
+begin
+  f(x);
+end.
--- /dev/null
+++ b/test/wrong_ref6.p
@@ -0,0 +1,16 @@
+var x: integer;
+
+proc g();
+begin
+  print_num(x); newline()
+end;
+
+proc f(var y: integer);
+begin
+  [ x := 1 : y := 3 ];
+  g()
+end;
+
+begin
+  f(x)
+end.
--- /dev/null
+++ b/test/wrong_refheap.p
@@ -0,0 +1,13 @@
+type ptr = pointer to integer;
+var p: ptr;
+
+proc f(var x: integer; p: ptr);
+begin
+  [ x := 5 : p^ := 3 ]
+end;
+
+begin
+  new(p);
+  f(p^, p);
+  print_num(p^); newline()
+end.
--- /dev/null
+++ b/test/wrong_refrarray.p
@@ -0,0 +1,11 @@
+var a: array 10 of integer;
+
+proc f(var x: integer);
+begin
+  [ a[3] := 5 : x := 3 ];
+  print_num(x); newline()
+end;
+
+begin
+  f(a[3])
+end.
--- /dev/null
+++ b/test/wrong_refrecord.p
@@ -0,0 +1,12 @@
+type point = record x, y: integer end;
+var p: point;
+
+proc f(var q: point);
+begin
```

```
+  [ p.x := 5 : q.x := 3 ];
+  print_num(p.x); newline()
+end;
+
+begin
+  f(p)
+end.
--- /dev/null
+++ b/test/wrong_refrecord2.p
@@ -0,0 +1,12 @@
+type point = record x, y: integer end;
+var p: point;
+
+proc f(var x: integer);
+begin
+  [ p.x := 5 : x := 3 ];
+  print_num(x); newline()
+end;
+
+begin
+  f(p.x)
+end.
--- /dev/null
+++ b/test/wrong_refrecord3.p
@@ -0,0 +1,13 @@
+type point = record x, y: integer end;
+var p: point;
+
+proc f(var x: integer);
+  var a: integer;
+begin
+  [ p.x := 5 : a := x ];
+  print_num(a); newline()
+end;
+
+begin
+  f(p.x)
+end.
--- /dev/null
+++ b/test/wrong_return.p
@@ -0,0 +1,8 @@
+proc f(x: integer): integer;
+begin
+  [ return x : return 7 ]
+end;
+
+begin
+  print_num(f(6)); newline()
+end.
--- /dev/null
+++ b/test/wrong_return2.p
@@ -0,0 +1,11 @@
+var x : integer;
+
+proc f(): integer;
+begin
+  [ return 4 : x := 7 ]
+end;
+
+begin
+  print_num(f()); newline();
+  print_num(x); newline()
+end.
--- /dev/null
+++ b/test/wrong_return3.p
```

```
@@ -0,0 +1,17 @@
+var y: integer;
+
+proc f(x: integer);
+begin
+  [
+    if x <= 0 then return end
+  :
+    y := y * 2
+  ];
+  print_num(x); newline();
+  print_num(y); newline();
+  f(x - 1)
+end;
+
+begin
+  f(12)
+end.
--- /dev/null
+++ b/test/wrong_while.p
@@ -0,0 +1,11 @@
+var x: integer;
+var y: integer;
+begin
+  [
+    while x > 3 do x := x - 1 end
+  :
+    y := x + 2
+  ];
+  print_num(x); newline();
+  print_num(y); newline()
+end.
```